File system virtual appliances

MICHAEL ABD-EL-MALEK

May 2010

CMU-PDL-09-109

Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Thesis committee

Prof. Gregory R. Ganger, Co-Chair (Carnegie Mellon University)
Prof. Michael K. Reiter, Co-Chair (University of North Carolina at Chapel Hill)
Prof. Garth A. Gibson (Carnegie Mellon University)
Dr. Orran Krieger (VMware)

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
MAY 2010		**00-00-2010 to 00-00-2010**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
File system virtual appliances	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,Dept. of Electrical and Computer Engineering,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Implementing and maintaining le systems is painful. OS functionality is notoriously di cult to develop and debug, and le systems are more so than most because of their size and interactions with other OS components. In-kernel le systems must adhere to a large number of internal OS interfaces. Though di cult during initial le system development, these dependencies particularly complicate porting a le system to di erent OSs or even across OS versions. This dissertation describes an architecture that addresses the le system portability problem. Virtual machines are used to decouple the OS on which a le system runs from the OS on which user applications run. The le system is distributed as a le system virtual appliance (FSVA), a virtual machine running the le system developers' preferred OS (version). Users runs their applications in a separate virtual machine, using their preferred OS (version). An FSVA design and implementation is described that maintains le system semantics with few, if any, code changes. This is achieved by sending all le system operations from the user OS to the FSVA. A uni ed bu er cache is maintained by using shared memory between the user OS and FSVA and by letting the user OS control the FSVA's bu er cache size. Features such as resource isolation and security are maintained through a single FSVA-per- user-OS design. Virtual machine migration is supported by simultaneously migrating a user OS and FSVA(s), maintaining shared memory mappings and live migration's low downtime. Several case studies demonstrate FSVAs' e ectiveness in providing OS- independent le system implementations. Measurements show that FSVA overheads on di erent workloads vary from 0{40%. The main overhead source is the communication latency between the user OS and FSVA. If a processor core is dedicated to an FSVA, a power-e cient polling mechanism reduces the overheads to 0{10%. Alternatively, relaxing the FSVA design goals by handling the frequent access-control le system checks in the user OS leads to similar overhead reductions as polling, but without the need for an additional core.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**126**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

To my family.

Abstract

Implementing and maintaining file systems is painful. OS functionality is notoriously difficult to develop and debug, and file systems are more so than most because of their size and interactions with other OS components. In-kernel file systems must adhere to a large number of internal OS interfaces. Though difficult during initial file system development, these dependencies particularly complicate porting a file system to different OSs or even across OS versions.

This dissertation describes an architecture that addresses the file system portability problem. Virtual machines are used to decouple the OS on which a file system runs from the OS on which user applications run. The file system is distributed as a *file system virtual appliance* (FSVA), a virtual machine running the file system developers' preferred OS (version). Users runs their applications in a separate virtual machine, using their preferred OS (version).

An FSVA design and implementation is described that maintains file system semantics with few, if any, code changes. This is achieved by sending all file system operations from the user OS to the FSVA. A unified buffer cache is maintained by using shared memory between the user OS and FSVA and by letting the user OS control the FSVA's buffer cache size. Features such as resource isolation and security are maintained through a single FSVA-per-user-OS design. Virtual machine migration is supported by simultaneously migrating a user OS and FSVA(s), maintaining shared memory mappings and live migration's low downtime.

Several case studies demonstrate FSVAs' effectiveness in providing OS-independent file system implementations. Measurements show that FSVA overheads on different workloads vary from 0–40%. The main overhead source is the communication latency between the user OS and FSVA. If a processor core is dedicated to an FSVA, a power-efficient polling mechanism reduces the overheads to 0–10%. Alternatively, relaxing the FSVA design goals by handling the frequent access-control file system checks in the user OS leads to similar overhead reductions as polling, but without the need for an additional core.

Acknowledgements

I had a great time in graduate school. This was in large part due to the wonderful people that I was fortunate enough to interact with.

First and foremost, I cannot thank my advisors Greg Ganger and Mike Reiter enough. I was lucky to have not one, but two advisors that cared about my development and research interests, always made time for me, and were always supportive. They taught me how to identify interesting problems, develop a solution, evaluate it, and clearly describe it. I enjoyed our interactions tremendously. I also owe all my knowledge of college basketball and football to them.

My dissertation grew out of Garth Gibson's idea of running a file system in a virtual machine. I thank him for his guidance during this work. Orran Krieger had many useful insights, encouraging me to view this work as leveraging multicores and motivating virtual machine-based microkernels.

I was fortunate to work with many great graduate students. Jay Wylie and Garth Goodson patiently explained distributed systems to me, and were wonderful mentors during my first two years. Jay continued informally mentoring even after finishing graduate school, for which I am very grateful. Matthew Wachs was instrumental to my dissertation: he was the primary designer and implementor of the unified buffer cache and migration support. I thank Jim Cipar for his contributions to FSVAs. Karan Sanghi implemented the NetBSD port in an impressively short time. The Self-* project was a great learning experience. I especially enjoyed working with Eno Thereska: our numerous collaborations were fun and Eno always kept focus on the research problems. I also enjoyed working with John Strunk, Mike Mesnier,

Chuck Cranor, James Hendricks, Shafeeq Sinnamohideen, Raja Sambasivan, and Andrew Klosterman.

The Parallel Data Lab was a stimulating environment. The annual retreats and visit days provided a forum with top industry representatives who provided feedback, interest, and support. I thank Greg Ganger, Garth Gibson, Bill Courtright and David Nagle for starting and maintaining this wonderful lab. Karen Lindenfelser, Linda Whipkey, and Joan Digney made the PDL events flow smoothly, and always provided good conversation. Bill Courtright had a lot of sage advise on research, industry, startups, and dealing with people.

My friends in Pittsburgh made life more fun, enriching, and educational. My first friend in Pittsburgh was Jon: although we significantly differed in our athletic abilities and shisha fondness, Jon was a fellow Sharp Edge and scotch aficionado and a great roommate for three years. My close Arab friends – Hanadie, Mansour, Nick, and Sarah – had a large impact on me and formed some of my closest friendships. Ashraf, Ippo and Neil made life much more fun. Mike Merideth introduced me to fine scotch, hi-fi speakers, and piano concertos – I, but not my wallet, will always be grateful. My last two years in Pittsburgh were full of pleasant times with Fayyad, Selen and the Greeks: Panickos, Kiki, Socrates, Panos and Michael. Finally, all of this would not have been possible if Shreyas Sundaram had not persistently convinced me to apply to CMU – hopefully he will absolve me of my firstborn promise.

Last, but not least, my family has always been supportive. Each of them encouraged me and made all of this possible, in their own special way. I dedicate this dissertation to them as a small token of appreciation.

Thanks to Chris Behanna (Panasas), Derrick Brashear (OpenAFS), Nitin Gupta (Panasas), Roger Haskin (GPFS), Sam Lang (PVFS), Rob Ross (PVFS), and Brent Welch (Panasas) for sharing their file system development experience and many anecdotes. Thanks to Ben Pfaff (POFS), Mark Williamson (XenFS), and Xin Zhao (VNFS) for sharing their code, which helped me quickly start the FSVA prototype implementation. Adam Pennington provided access to his AFS server. Gregg Economou, Michael Stroucken,

and Doug Needham installed and maintaining the PDL's excellent computing infrastructure.

Thanks to the members and companies of the CyLab Corporate Partners and the PDL Consortium (including APC, Data Domain, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, IBM, Intel, LSI, Microsoft Research, NetApp, Oracle, Seagate, Sun Microsystems, Symantec, and VMware) for their interest, insights, feedback, and support. This material is based on research sponsored in part by the National Science Foundation, via grants CNS-0326453 and CCF-0621499, by the Department of Energy, under Award Number DE-FC02-06ER25767, and by the Army Research Office, under agreement number DAAD19–02–1–0389. Intel and Network Appliance donated hardware donations that enabled this work.

Contents

Figures

Tables

1 Introduction

1.1 The problem

Implementing and maintaining file systems is painful. OS functionality is notoriously difficult to develop and debug, and file systems are more so than most because of their size and interactions with other OS components. In-kernel file systems must adhere to the OS's virtual file system (VFS) interface [52], but that is the easy part. File system implementations also depend on a large number of internal OS interfaces. For example, a file system developer must understand the memory allocation, caching, threading, locking/preemption, networking (for distributed file systems), and device access (for local file systems) interfaces and semantics. In particular, correctly handling locking and preemption is notoriously difficult.

Though difficult during initial file system development, these dependencies particularly complicate porting a file system to different OSs or even OS versions. While VFS interfaces vary slightly across OSs, the other OS internal interfaces greatly vary, making porting of file systems painful and effort-intensive.

In practice, these portability issues require substantial developer effort — approximately 50% of the effort, in the estimate of some developers (§2.3.3). In many cases, the porting cost is unjustified, and file system developers simply forgo OSs that pose too large a hurdle. File system users are then either forced to change OS (versions) or switch to a file system that is less suitable to their needs but available in their preferred OS (version).

1.2 Thesis statement

This dissertation proposes a new approach for OS-independent file system implementations. Specifically, the thesis statement is:

Executing a file system in its own virtual machine enables OS-independent file system implementations by decoupling the file system OS from the user OS. This decoupling can be efficiently achieved while maintaining file system, OS, and virtualization features with few changes to these components.

This dissertation validates the thesis in the following manner:

(1) We describe an architecture that enables OS-independent file system implementations. We present a design, following this architecture, that maintains file system, OS, and virtualization features.

(2) We built a working prototype of the design, confirming the feasibility of the architecture and design.

(3) We present case studies that demonstrate the prototype's effectiveness in providing OS-independent file system implementations. We demonstrate that this can be done efficiently through performance measurements. We analyze the code changes to demonstrate the small size of the changes to the file system, OS, and virtual machine monitor.

1.3 Dissertation overview

This dissertation offers a new approach for OS-independent file system implementations, leveraging virtual machines (VMs) to decouple the OS on which the file system runs from the OS on which the user applications run. The file system is distributed as a *file system virtual appliance* (FSVA), a pre-packaged virtual appliance [87] loaded with the file system. The FSVA runs the file system developers' preferred OS (version), with which they have performed extensive testing and tuning. Users runs their applications in a

separate VM, using their preferred OS (version). File system-agnostic proxies in both OSs efficiently pass VFS operations from the user OS to the FSVA and maintain OS and virtualization features.

Since it runs in a distinct VM, the file system can be used by users who choose OSs to which it is never ported. The file system is isolated from both kernel- and user-space differences in user OSs, because it interacts with just the single FSVA OS version. The result is that users are free to select any OS of their choice, and file system developers support only the single FSVA OS. Furthermore, with appropriate design, this architecture can leverage unmodified legacy file system implementations.

For the FSVA approach to work, the file system-agnostic proxies must be a "native" part of the OS — they must be maintained across versions by the OS implementers. The hope is that, because of their small size and value to a broad range of file system users and developers, OS vendors would be willing to adopt such a proxy.

1.4 Contributions

This dissertation makes the following contributions:

(1) We propose the FSVA architecture as a solution to the file system portability problem.

(2) We describe an FSVA design and prototype, implemented for Linux and NetBSD, that maintains file system, OS, and virtualization features with few changes to these components. We describe and evaluate a number of performance optimizations that are necessary for reasonable performance, and evaluate the prototype's performance.

(3) We elucidate the design space when separately executing a file system in a VM. Others have also done so for a variety of reasons (e.g., for security or to provide virtualization-optimized file systems). We analyze the major design decisions, discuss common challenges and solutions (e.g., a unified buffer cache), and highlight common correctness and performance pitfalls.

(4) We analyze the sources of latency in traditional inter-VM communication techniques and present a novel energy- and performance-efficient mechanism.

1.5 Outline

The remainder of this dissertation is organized as follows. Chapter 2 describes the challenges in porting file systems, current approaches, and related work. Chapter 3 gives an overview of the FSVA architecture, describes enabling technology trends, discusses the architecture's viability, and concludes with the architecture's costs and limitations. Chapter 4 overviews the FSVA design space, lists our design goals, and describes an FSVA design that satisfies those goals. Chapter 5 details our prototype implementation. Chapter 6 evaluates the prototype along two dimensions. First, using several case studies, we demonstrate FSVAs' effectiveness in providing OS-independent file system implementations. Second, we analyze the FSVA performance overhead through micro- and macro-benchmarks, and study the efficacy of various optimizations. Chapter 7 describes our experience in implementing the prototype for different OSs and virtual machine monitors and provides lessons for others interested in moving the file system into a dedicated virtual machine. Chapter 8 concludes and discusses possible feature work.

2 Background and related work

The goal of this dissertation is to enable OS-independent file system implementations. This chapter explains why this is a worthwhile goal and previous attempts to achieve it. We describe the OS-file system interface, discuss the difficulties in developing and porting file systems, and provide anecdotal experience from file system developers on the difficulties of porting file systems. Existing approaches and their shortcomings are then described. This chapter ends with a discussion of related work.

2.1 Terminology

Before proceeding, it is necessary to clarify my usage of a few terms.

The term *file system* is commonly used to refer to data as well as to the software for accessing and manipulating the data. This dissertation is concerned with the latter meaning. For brevity, we use *file system* to refer to file system software. The terms *file system*, *file system software*, and *file system implementation* are used interchangeably.

The Unix file system layer has two components: the per-file *vnode* and per-file system *virtual file system* (VFS) layers. A file system must adhere to both of these interfaces. For brevity, we will collectively refer to both interfaces as the *VFS interface*.

The FSVA architecture enables users to access a file system written for a different OS or a different OS version. To avoid the cumbersome *different OS or OS version* or *different OS (version)* phrases, we use *different OS* to encompass both meanings.

2.2 OS structure and file system implementations

File system implementations must adhere to internal OS interfaces. This section provides an overview of the OS-file system interfaces, their design goals, and their effect on file system development.

File system implementation structure in modern OSs is based on Sun Microsystems' virtual file system (VFS) architecture [52]. This architecture was created to support multiple file system implementations in an OS. Specifically, Sun Microsystems' development of the Network File System (NFS) [69, 84] necessitated kernel changes to simultaneously accommodate NFS with the local filesystem. Figure 2.1 recreates the VFS architecture block diagram from the VFS paper [52], illustrating the relationship between file system implementations and other OS components. File system operations, originating from either system calls or kernel operations, are sent to the generic VFS layer. For some operations, the VFS layer can respond without involving the file system. Otherwise, the VFS layer passes control to the file system through previously-registered VFS callbacks. The set of all VFS callbacks forms the *VFS interface*.

The primary goal of the VFS architecture was to "accommodat[e] multiple file system implementations within the Sun UNIX kernel" [52]. This was achieved through two techniques. First, outside the file system layer, references to specific file system implementations were replaced by references to the generic VFS layer. Second, in the file system layer, the VFS interface provided a well-defined interface to file system implementations. In Figure 2.1, these techniques correspond to narrow *FS system calls* and *VFS callbacks* interfaces. The result is a clean encapsulation of file system implementations, from the kernel's perspective. The VFS paper concludes that the VFS layer "has been proven to provide a clean, well defined interface *to* different file system implementations" [emphasis added] [52].

But, the interface *from* file system implementations to the rest of the kernel was not well-defined. Although the VFS paper's illustration shows a thin interface from file systems to the rest of the kernel (Figure 2.1), the reality is much more complicated. File systems rely on and conform to many

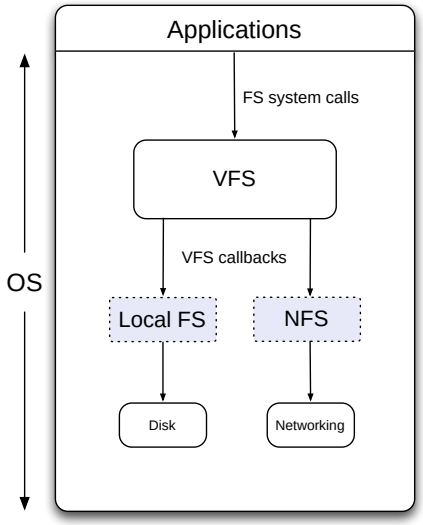


Figure 2.1. Recreation of the original VFS architecture figure.

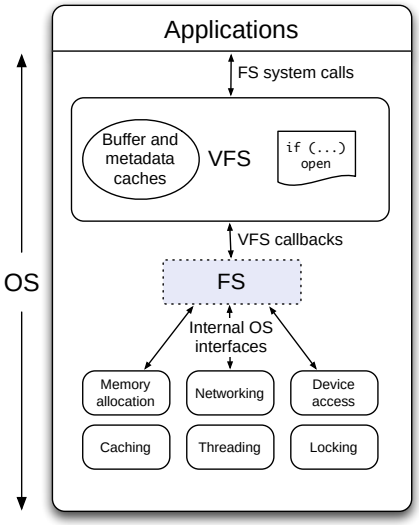


Figure 2.2. VFS architecture with explicit interfaces.

internal OS interfaces and semantics: memory allocation, paging (for memory mapping), locking, preemption, threading, networking (for distributed file systems), and device access (for local file systems) interfaces and semantics. Additionally, file systems depend on internal VFS interfaces, such as common VFS helper functions and the buffer, inode and directory entry caches [62]. Figure 2.2 expands the file systems' dependencies to the rest of the kernel.

The large number of dependencies on internal OS interfaces complicates file system development. To see why, consider the two aspects of interfaces: syntax and semantics. Quick code inspection and compiler messages aid in adhering to interfaces' syntax. In contrast, understanding and adhering to interfaces' semantics is much more challenging. For example, correctly handling kernel locking and preemption is notoriously difficult.

Of course, file system implementations are not unique in their dependency on internal OS interfaces. Device drivers are also equally affected. But, file systems have much greater interdependencies with memory management than device drivers. Also, there are many more interactions and a much richer interface. This is a result of file systems' aggressiveness in hiding the

I/O subsystem's latency. File systems perform extensive caching (e.g., the buffer, inode and directory entry caches) and frequently use asynchronous operations (e.g., writeback, distributed file system callbacks).

In summary, file system development is complicated by the dependency on a large number of internal OS interfaces.

2.3 The problem: porting file systems

Based on user demand, file system developers must port their file system to different OSs. This section describes the difficulty of porting file system implementations, discusses three manifestations of this problem, and provides anecdotal evidence from file system developers.

2.3.1 Why porting is difficult

As described in the previous section, file system implementations are tightly intertwined with internal OS interfaces. The lack of standardization among internal OS interfaces significantly complicates porting. Although VFS interfaces vary slightly across OSs, the other internal OS interfaces greatly vary, making file system porting painful and effort-intensive.

From VFS's introduction, different VFS-like interfaces existed for each OS. In the 1986 USENIX Summer Conference, Sun's VFS [52] was followed by AT&T's File System Switch [78], which was followed by Digital Equipment's Generic File System [80]. In Sun's VFS paper, Kleiman stated that "Sun is currently discussing with AT&T and Berkeley the merging of this interface with AT&T's File System Switch technology. The goal is to produce a standard UNIX file system interface." [52] Later that year, Karels and McKusick proposed a common filesystem interface [50]. These proposals were never adopted. Different OSs continue to have different VFS interfaces.

But, in addition to the VFS interface, file systems depend on a myriad of internal OS interfaces, as described in §2.2. These aspects vary widely across OSs, and they often vary even across versions of the same OS. Adapting to such variation is the primary challenge in porting file system implementations.

Again, this problem has long been known. In a 1993 paper advocating user-level file systems, Weber enumerated many differences among different OSs: locking, read-ahead, syntax, caching (unified versus separate buffer and page caches), kernel preemptiveness, semantics of per-process read/write atomicity, threading, memory allocation, networking, and device access [106]. Weber notes that "from a third party point of view there are two major problems: few vendors have the same VFS interface and few vendors provide release-to-release source or binary compatibility for VFS modules." Most tellingly, he later observes that "the problem is not that any particular VFS implementation is especially complicated or difficult to understand, but that too many VFS implementations exist." [106]

Eight years later, the problem still persisted. In a 2001 paper advocating user-level file systems, Mazieres describes the difficulty of kernel file system development: "Developing new Unix file systems has long been a difficult task. The internal kernel API for file systems varies significantly between versions of the operating system, making portability nearly impossible. The locking discipline on file system data structures is hair-raising for the non-expert." [61]

In §2.2, we distinguished between two aspects of interfaces: syntax and semantics. This distinction is helpful in understanding part of the difficulty of porting file systems. Syntactic differences among internal OS interfaces can be wrapped inside a file system's compatibility layer. For example, the compatibility layer can provide an abstract memory allocator function, hiding the syntax of each OS's memory allocator. Syntactic abstraction simplifies file system porting. By isolating OS-specific interfaces to a fraction of the file system codebase, the porting cost is reduced to being proportional to the size of the small compatibility layer, not to the size of the entire file system.

Unfortunately, semantic differences are often not as amenable to abstraction. Consider kernel preemptiveness. Porting a file system from a non-preemptible kernel to a preemptible kernel has wide ramifications to the file system-wide locking discipline. Thus, unlike syntactic differences, the porting cost in handling inter-OS semantic differences is proportional to the entire file system size. Additional examples are in §2.3.3.

2.3.2 Problem manifestation

Porting file systems from one OS to another is difficult, whether the second OS is a different OS or a different version of the same OS. The relationship between the two OSs leads to three manifestations of the portability problem.

Inter-OS porting is characterized by porting a file system from one OS to a different OS. Porting NetBSD's log-structured file system (LFS) to Linux is an example.

A less-appreciated file system porting challenge is dealing with *intra-OS version porting*. Even when the VFS interface remains constant, internal file system compatibility rarely exists between one kernel version and the next. The same issues that plague inter-OS porting also affect intra-OS version porting. Changes in syntax, locking semantics, memory management, and preemption practices create differences that require OS version-specific code in the file system implementation.

Intra-OS version porting takes two forms. In *forward porting*, a file system developed for one OS version requires modifications to function in each subsequent version of that OS. For "native" file systems supported by the kernel implementers (e.g., ext2 and NFS in Linux), appropriate corrections are made in the file system as a part of the new kernel version. For third-party file systems, however, they are not. As each new kernel version is released, whether as a patch or a complete replacement, the third-party file system maintainers must figure out what changed, modify their code accordingly, and provide the new file system version. Because users of the third-party file systems may be using any of the previously supported OS versions, all must be maintained and the code becomes riddled with version-specific `#ifdef`s, making it increasingly difficult to understand and modify correctly.

Backward porting is the second intra-OS version porting form, in which a file system developed for the latest OS version must be backported to support users of a previous OS version who cannot upgrade to the latest OS version. File system developers performing backward porting face identical issues as in forward porting. As an example of backward porting, some Linux vendors have backported ext4, a file system introduced in the 2.6.28 kernel,

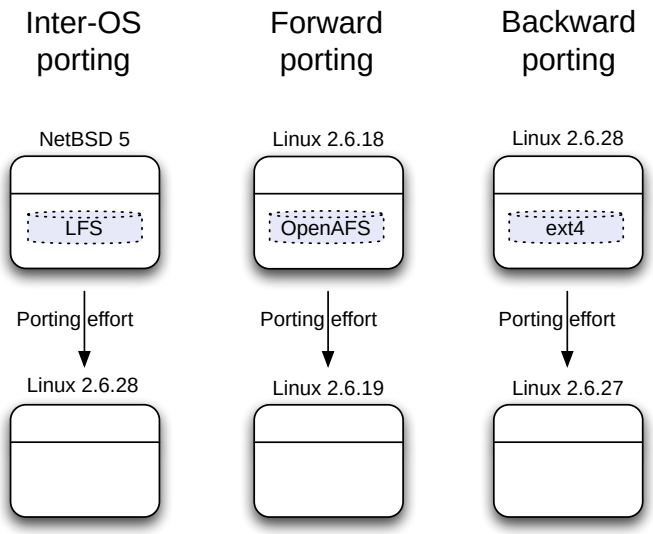


Figure 2.3. Three manifestations of the portability problem.

to earlier Linux kernel versions. But, due to the high cost of porting, this was not comprehensively performed for all prior Linux kernel versions.

Figure 2.3 illustrates the three manifestations of the file system portability problem. Weber used the terms "VFS portability" and "lock-step release" to refer to the first two forms [106], respectively, while Skinner and Wong called them the "compatibility and stability" problems [92]. For brevity, we will simply use "file system portability" to refer to all three manifestations, and "different OS" to encompass different OSs as well as different versions of the same OS.

2.3.3 Anecdotal experiences

To better understand the file system portability problem, we interviewed developers of four third-party file systems: GPFS [88], OpenAFS (an open-source implementation of AFS [44]), Panasas DirectFLOW [108], and PVFS [16]. All four file systems have been widely deployed for many years. Because the inter-OS porting problem is well-known [50, 61, 92, 106], and PVFS and Panasas DirectFLOW are only available on Linux, we focus on the developers'

Type	Description
VFS interface	The vector I/O `readv` and `writev` VFS callbacks were replaced with the asynchronous I/O `aio_read` and `aio_write` callbacks (2.6.19). `sendfile` was replaced by `splice` (2.6.23).
Virtual memory	The interface for the virtual memory page fault handlers, overridable by a file system, was changed (2.6.23).
Caching	The parameters for the kernel cache structure constructors and destructors were changed (2.6.20).
Structures	The per-inode `blksize` field was removed (2.6.19). The process task structure no longer contained the thread pointer (2.6.22).
Header files	`config.h` was removed (2.6.19).

Table 2.1. Examples of interface syntax changes.

experiences with intra-OS porting. Naturally, developers performing inter-OS porting face all of these issues and more. The file systems are all distributed file systems; the developers describe their experience in maintaining the Linux client-side code.

Interface syntax changes. The first changes that a file system developer encounters in an OS update are interface syntax changes, due to compilation errors. Table 2.1 contains a representative list, with the corresponding Linux kernel version in parentheses. Some examples were conveyed by the developers, and others were gleaned from looking at OpenAFS's and PVFS' logs.

Although some of these changes may seem trivial, they are time-consuming and riddle source code with version-specific `#ifdef`s that complicate code understanding and maintenance. Furthermore, *every* third-party file system team must deal with each problem as it occurs. Examination of the open-source OpenAFS and PVFS change logs shows that both file systems contain fixes for each of these (and many similar) issues.

Policy and semantic changes. Even if interfaces' syntax remain constant across OS releases, implementation differences can have subtle effects that are

difficult to debug. Table 2.2 lists examples of policy and semantic changes.

Because the policy and semantic changes were not documented, each third-party file system team had to discover them through kernel debugging and code analysis, and then work around them. Many of these bugs (e.g., Write-back, Stack Size, and Locking) manifest themselves much later than the offending code, greatly complicating debugging.

Overall statistics. To appreciate the magnitude of the problem, consider the following statistics. Panasas' Linux portability layer supports over 300 configurations.[1] PVFS developers estimate that 50% of their maintenance effort is spent dealing with Linux kernel issues [54]. The most frequently revised file in the OpenAFS client source code is the Linux VFS-interfacing file [71]. An OpenAFS developer estimates that 40% of Linux kernel releases necessitate an updated OpenAFS release [12].

One may be tempted to brush off the preceding difficulties as Linux-only anomalies. But, although most pronounced for Linux, with its independent and decentralized development process, this problem poses challenges for file system developers targeting any OS. Furthermore, given Linux's wide deployment in the server marketplace, this is a real problem faced by third-party file system developers, as the statistics demonstrate — simply dismissing it is inappropriate. Finally, these same porting issues are experienced during inter-OS porting, and a solution that addresses the full file system portability problem would be attractive.

2.4 Current approaches

User-level file systems. Most OS vendors maintain binary compatibility for user-level applications across OS releases. As a result, user-level file systems have been proposed as a solution to the intra-OS porting problem [8, 61, 106]. Additionally, by only using the standard POSIX programming interfaces [30],

[1] Due to differences among distributions and processor types, Panasas clusters Linux platforms by a <distribution name, distribution version, processor architecture> tuple. Currently, Panasas supports 45 Linux platforms. In addition, within each platform, Panasas has a separate port for each kernel version. The result is over 300 configurations.

Type	Description
Memory Pressure	Some RedHat Enterprise Linux 3 kernels are not robust during low memory situations. In particular, the kernels can block during allocation despite the allocation flags specifying no blocking. This results in minutes-long delays in dirty data writeback under low memory situations. RedHat acknowledged the semantic mismatch but did not fix the issue [77]. A file system vendor was forced to work around the bug by carefully controlling the number of dirty pages (via per-kernel-version parameters) and I/O sizes to the data server (thereby negatively impacting server scalability).
Write-back	Linux uses a write-back control data structure (WBCDS) to identify dirty pages that need to be written to stable storage. A file system populates this data structure and passes it to the generic VFS layer. Linux 2.6.18 changed the handling of a sparsely-initialized WBCDS, such that only a single page of a specified range was actually written. This caused a file system to mistakenly assume that all pages were written, resulting in data corruption.
Stack Size	RedHat Enterprise Linux kernels often use a smaller kernel stack size (4 K instead of the default 8 K). To avoid stack overflow, once this was discovered, a file system vendor used continuations to pass request state across kernel threads. But, continuations are cumbersome for developers and complicate debugging. This illustrates how one supported OS's idiosyncrasies can complicate the entire file system, not just the OS-specific compatibility layer.
Locking	Accessing existing inode fields required the inode lock to be held, whereas previously no locking was required.
Radix Tree	The Linux kernel provides a radix tree library. The 2.6.20 kernel required the least significant bit of stored values be 0, breaking a file system that stored arbitrary integers.

Table 2.2. Examples of policy and semantic changes.

file system developers can, in theory, simply recompile their unmodified file system to address inter-OS portability.

User-level file systems are implemented either through a small kernel module that reflects file system calls into user-space [8, 35, 48, 106, 109] or through a loopback NFS or CIFS server that leverages existing kernel NFS or CIFS client support [4, 15, 17, 39, 61]. Figures 2.4 and 2.5 illustrate both approaches, respectively. The interface between the kernel and user-space file system affects the file system's portability and semantics. Using a widely-supported distributed file system protocol, such as NFS, avoids the need for additional kernel code. But, this limits file systems' semantics by the information (e.g., NFS lacks `close` callbacks) and control (e.g., NFS's weak cache consistency) available to them.

User-level file systems are not sufficient, for several reasons. First, user-level file systems are unable to accommodate existing kernel-level file system implementations. Second, user-level file systems still depend on the kernel to provide low-level services such as device access, networking, and memory management. Changes to the behavior of these components can still affect a user-level file system. For instance, Table 2.2's Memory Pressure example would not be solved by user-level file systems. Thus, user-level file systems are not fully isolated from the underlying OS components.

Third, user-level file systems can deadlock because most OSs were not designed to robustly support a user-level file system in low-memory situations [61]. Such deadlocks can be avoided by using a purely event-driven structure, as the SFS toolkit does [61], but at the cost of restricting implementer flexibility. Fourth, user-level file systems provide no assistance with user-space differences, such as shared library availability and OS configuration file formats and locations, or the use of non-portable OS interfaces (e.g., remote memory, as in DAFS [26]).

Despite their disadvantages, user-level file systems are sometimes useful. They permit quick prototyping and are sufficient in situations where full file system robustness or portability are not critical. But, user-level file systems are not a general solution.

Rump allows the execution of unmodified NetBSD kernel file systems in

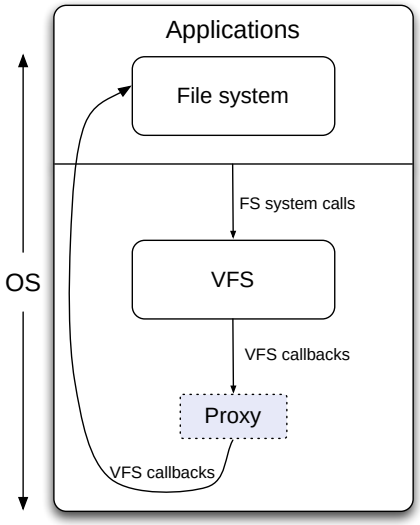


Figure 2.4. User-level file systems via kernel proxy.

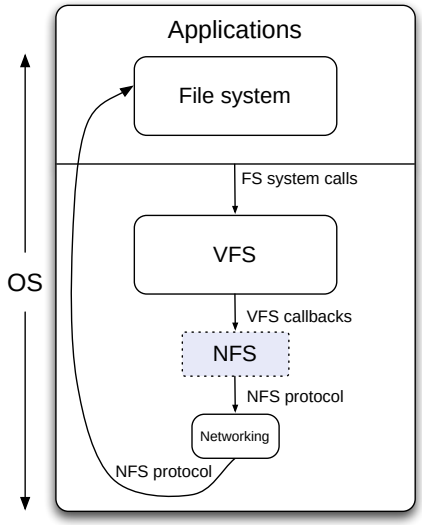


Figure 2.5. User-level file systems via NFS loopback.

user-space, by reimplementing the necessary internal OS interfaces in user-space [49]. This was previously performed by Thekkath et al. to accurately model storage system performance [99], and was suggested by Yang et al. but considered too burdensome [110]. Because this approach essentially adds a library on top of existing user-space support, it suffers from all but the first of the user-level file system deficiencies. Furthermore, although rump accommodates unmodified kernel-level file systems, it does so at a cost: reimplementing the internal OS interfaces that file systems rely on. These interfaces are much broader than the VFS interface. To achieve inter-OS operability, this reimplementation must be performed for each OS (version). Also, conflicts between kernel and user-space interfaces can pose problems. For example, their NFS server required modification due to conflicts between the kernel and user-space RPC portmapper and NFS mount protocol daemon.

Language-based approaches. FiST provides a specialized language for file system developers [111]. The FiST compiler generates OS-specific kernel modules. Given detailed information about all relevant in-kernel interfaces, updated for each OS version, FiST could address inter- and intra-OS syntax

changes. But, FiST was not designed to offer assistance with policy and semantic changes. These changes can require substantial file system revision (e.g., Table 2.2's Stack Size example) and hence cannot be simply isolated from a file system implementation. Also, a specialized language is unlikely to be adopted unless it is expressive enough to address all desirable control. This is far from a solved problem. Furthermore, FiST does not accommodate existing file system implementations.

Selective availability. Due to the inadequacies of the preceding approaches, file systems developers fall back on what we refer to as *selective availability*. They select particular OS versions to support, use brute force to port their file system to these OS versions, and simply avoid porting their file system to other OSs or OS versions. Thus, the file system portability problem results in a large barrier for those seeking to innovate and wears on those who choose to do so.

The four file system developers interviewed in §2.3.3 practice selective availability, to varying extents. Panasas DirectFLOW and PVFS are available only on Linux, and supported on a subset of Linux kernel versions. GPFS is available on Linux only for some kernel versions. Users suffer from selective availability. For example, Argonne National Laboratory uses GPFS and PVFS for home directories and scientific data, respectively. Selective availability forces Argonne to use an old Linux kernel that is supported by *both* file systems [83]. This causes pain to developers, as they are unable to use newer Linux features.

2.5 Additional related work

File systems and VMs. Several research projects have explored running a file system in a VM, for a variety of reasons such as extensibility, sharing, performance, and security. POFS [76] proposes that virtual machine monitors should provide a higher-level file system interface to a VM, instead of the traditional device-like block interface, in order to gain sharing, security, modularity, and extensibility benefits. XenFS [58] shares a file system VM among multiple user VMs, in order to share a buffer cache and provide a single

copy-on-write file system image. VNFS [112] optimizes NFS performance when an NFS client is physically co-located with an NFS server, using shared memory to enable zero-copy data movement and to allow clients to directly read the NFS server's metadata. VPFS [107] builds a trusted storage facility out of untrusted legacy file systems using microkernels, and Matthews et al. [59] protect user data in the event of security attacks by storing the data in an NFS server virtual appliance. VMware Workstation [101] provides a "Shared Folders" feature to enable a guest VM to access the host OS's file system.

The FSVA architecture adapts these ideas to address the file system portability problem. But, the differing goals lead to different design decisions. First, user OSs cannot cache data or metadata, since FSVAs are file system-agnostic conduits to *existing* file system implementations. To maintain file system semantics, all user OS VFS calls must be sent to the FSVA. Second, separate FSVAs are employed for each user VM, to maintain virtualization features, such as migration and resource accounting, whereas others focus on using a single file system VM per physical machine to increase efficiency.

FSVAs also maintain OS features such as a unified buffer cache and VM features such as migration. Providing such features is orthogonal to the particular client caching and sharing design decisions; other systems can benefit from the FSVA solution to these problems.

OS structure. The FSVA architecture is an application of microkernel concepts [2, 42]. Microkernels execute OS components in separate servers. Doing so allows independent development and flexibility. But, traditional microkernels require significant changes to OS structure. FSVAs leverage virtualization to avoid the upfront implementation costs that held back microkernels.

LeVasseur et al. [57] reuse existing device drivers in different OSs by running them in a VM. Nooks [98] increases the reliability of commodity OSs while reusing existing drivers through lightweight kernel protection domains. Soft devices [104] simplify device-level development by reusing Xen's narrow paravirtualized device interface. FSVAs share these approaches' aim of leveraging existing kernel code and simplifying OS support or reliability.

In contrast, FSVAs deal with the richer file system interface while retaining OS and virtualization features.

Software engineering approaches. The software engineering community has studied the general problem of variability management. Software product lines [21] are a disciplined approach to finding and reusing common functionality (and interfaces) among related products. In a single vendor environment, or when multiple vendors agree on a common interface, this can be effective.

Unfortunately, different OS vendors (and even different releases of the same OS) have failed to agree on common and comprehensive internal OS interfaces. Different design choices and backward compatibility mean that the differences in OSs' internal interfaces are here to stay.

Language-based approaches. Padioleau et al. [74] characterize changes in Linux device drivers due to intra-OS interface changes. While they focus on device drivers, file systems face similar issues because both components rely on internal OS interfaces for memory allocation, locking, etc. Based on this study, Padioleau et al. then developed Coccinelle [73], a program transformation tool that automatically updates Linux device drivers after interface changes. While Coccinelle could handle some of the interface syntax changes that we described, like FiST [111], it would be unable to mitigate the policy and semantic changes. The latter require much more intrusive file system changes.

Fast IPC. Dean and Armand describe [25] how the Mach and CHORUS microkernels achieved high file system performance. Mach aggressively uses shared memory: clients have read-only access to the file system server memory, allowing common-case reads to proceed without a context switch to the file system server. CHORUS avoids Mach's shared memory usage, since it couples the file system client and server code. Instead, the file server is executed in supervisor mode to decrease the context switch overhead. Like CHORUS, the FSVA architecture avoids directly serving read operations in the user OS, in order to preserve file system semantics and remain file system-agnostic. Unlike CHORUS, FSVAs are not executed in supervisor mode, since the virtual machine monitor (VMM) is already using this privilage level.

User-level Remote Procedure Call [7] reduces local RPC overhead by avoiding protection boundary crossings. Processes enqueue requests and responses to a shared memory region and only invoke the kernel if the other process is not executing. FSVAs similarly avoid VMM scheduling and synchronous interrupts to achieve fast inter-VM event notification.

Fido [14] enables zero-copy inter-VM data movement through a single shared address space, in the spirit of single address-space operating systems [18]. FSVAs avoid data copies by using hypervisor shared memory hypercalls. Adopting Fido's single address-space approach would eliminate the need for the shared memory hypercalls.

3 Architecture

This chapter describes the FSVA architecture. First, we provide background on two technology trends that enable the architecture. Then, we give an overview of the FSVA architecture and discuss its viability. The chapter concludes with a discussion of the architecture's costs and limitations.

3.1 Technology trends

The FSVA architecture is enabled by two technology trends: virtualization and multicore processors. Virtualization is a fundamental building block for the FSVA architecture. Multicore processors enable high performance given current processor, OS, and virtual machine monitor architectures.

3.1.1 Virtualization

Virtualization is a technique for providing, and possibly sharing, a system or component interface that is potentially different from the underlying resources' interface [81, 93]. Depending on the resource, different types of virtualization are possible; for example, virtualization can occur at the hardware, storage, network, or process level. In this dissertation, we use *virtualization* to refer to hardware-level virtualization: the ability to concurrently execute multiple OSs on a physical machine [10]. Each OS executes in an isolated *virtual machine* (VM), observing a hardware interface that is usually identical to the underlying hardware interface [81].

Virtualization has a variety of uses. In the 1960s, IBM invented virtualization to time-share mainframes, allowing users to run isolated, and

possibly different, OSs [22, 38]. Interest in virtualization was revived in the late 1990s to serve a similar purpose: efficiently use large shared memory multiprocessors through simultaneous execution of multiple commodity OSs, thereby avoiding the need to develop scalable OSs [13]. VMware popularized the use of virtualization for application compatibility, program testing and development, data isolation, and server consolidation [81]. In addition to consolidation, virtualization is used in the enterprise for migration (enabling load-balancing and scheduled downtime), security sandboxing, fault tolerance [23], and disaster recovery [32, 63]. On the desktop, virtualization simplifies software distribution and maintenance [87] and is projected to be used on 660 million PCs by 2011 [36]. In high performance computing, virtualization has been proposed to increase developer productivity [45, 65].

Architecturally, there are two virtualization approaches: *native* (also known as *type 1*) or *hosted* (or *type 2*). In native virtualization, a *virtual machine monitor* (VMM) directly executes on hardware, multiplexing the physical resources among VMs. In hosted virtualization, a VMM runs as an ordinary process in a *host OS*, possibly with kernel-level extensions in the host OS. The host OS directly executes on hardware, and VMs execute inside the VMM process. Figures 3.1 and 3.2 illustrate the two architectures. The two approaches trade off performance, robustness, and ease of deployment.

3.1.2 Multicore processors

A substantial shift in microprocessor architecture has occurred due to physical limitations, such as heat dissipation, power consumption, and leakage currents [70]. Vendors, unable to sustain their previous rate of processor clock frequency increases, are instead adding more processing cores per circuit die, or chip. In contrast to traditional symmetric multiprocessing architectures, the multiple cores in a chip multiprocessor have faster interconnects and share caches [41].

The advent of multicore computing is disruptive for software. The majority of software applications are single-threaded and rely on the traditional increases in clock frequency for faster program execution on newer processors.

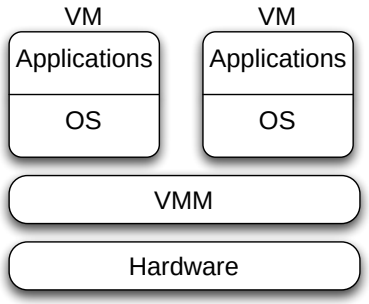


Figure 3.1. Virtualization using a native VMM.

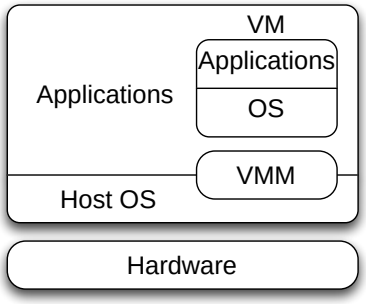


Figure 3.2. Virtualization using a hosted VMM.

But, multicores processors change this trend. With future processors unlikely to see substantial increases in clock frequency, user-perceived performance improvements must arise from rewriting software to make it parallel [97, 96]. Multithreading, the traditional parallel programming construct, is unlikely to be the solution, due to programming complexity [97]. Consequently, there is much research on alternative programming paradigms, such as transactional memory [43, 55].

Virtualization is likely to play a role in exploiting multicore processors. Safe and flexible resource sharing, provided by virtualization, can enable high utilization of multicore processors. Specifically, in contrast to OS-level resource multiplexing, virtualization's narrow interface ensures strict isolation between applications running in different VMs. Virtualization also allows different applications to use different OSs. Furthermore, virtualization eliminates the need to develop highly scalable OSs that exploit multicore parallelism.

3.2 Architecture overview

The FSVA architecture leverages virtualization to solve the file system portability problem. A file system implementation runs in a virtual machine (VM) executing the file system developer's preferred OS. We refer to the file system VM as a *File System Virtual Appliance* (FSVA). User applications

run in a user's preferred OS, possibly in a VM. File system-agnostic proxies in both OSs translate to/from a common VFS interface, using efficient VMM communication primitives. The proxies also maintain OS and virtualization features such as a unified buffer cache and migration, respectively. Figure 3.3 illustrates the FSVA architecture. The FSVA design space and a specific design point are explored in Chapter 4. This chapter discusses the generic, high-level FSVA architecture.

By decoupling of the user OS[1] from the file system OS, the FSVA architecture addresses the compatibility challenges discussed in §2.3. A file system developer implements his file system in a single OS version without concern for users' particular OSs. The file system is isolated from both kernel- and user-space differences in user OSs, because it interacts with just the single FSVA OS version. Policy and semantic issues like the poor handling of memory pressure and write-back (Table 2.1) can be addressed by simply not using such a kernel in the FSVA — the file system developer can choose an OS to suit the file system, rather than being forced to work with a user's chosen OS. Similarly, users are free to choose any OS (version). Thus, the FSVA approach handles all three forms of the file system portability problem (§2.3.2).

For the FSVA approach to work, the user OS and FSVA proxies must be a "native" part of the OS — they must be maintained across versions by the OS vendors. The hope is that, because of their small size and value to a broad range of file system users and developers, OS vendors would be willing to adopt such a proxy. FUSE [35], a proxy for user-level file systems, has been integrated into Linux, NetBSD, and OpenSolaris, and we envision a similar adoption path.

FSVAs do not preclude file system developers from porting their file systems to different OSs. Indeed, they might still do so to get new features, for improved performance, or for OS bug fixes. But, FSVAs enable such porting to occur at the developers' pace, not at the users' pace. Developers

[1]The file system running in the FSVA may be a client component of a distributed file system. To avoid client/server ambiguities, we use *user* and *FSVA* to refer to the file system user and VM executing the file system, respectively.

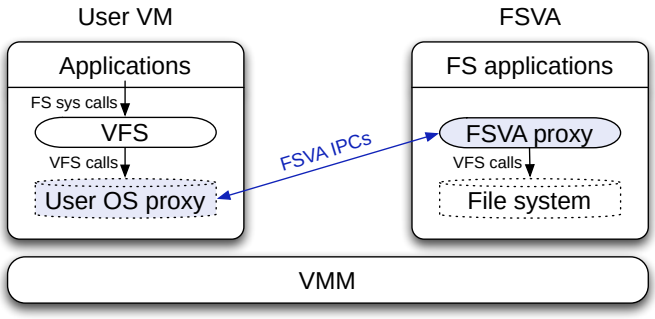


Figure 3.3. FSVA architecture. A file system and its (optional) management applications run in a dedicated VM. File system-agnostic proxy running in the user OS and FSVA pass VFS calls via an efficient inter-VM IPC layer.

can skip porting to most OSs and select a new stable OS (version) when desired.

The FSVA architecture borrows aspects from VFS, user-level file systems, microkernels, and virtual appliances. In the spirit of VFS, the FSVA interface isolates file systems from the user OS and vice versa, but only more so. Similar to user-level file systems, a small file system-agnostic proxy is maintained in the kernel. But, instead of a user-level process, the proxy allows the file system to be implemented in a dedicated VM. This leverages legacy file system implementations and provides stronger isolation from user OSs, overcoming the policy and semantic challenges described in §2.3. Like microkernels, FSVAs enable independent, flexible OS development. But, virtualization enables the *virtual appliance* software distribution model [87], avoiding microkernels' OS rearchitecture cost.

The FSVA architecture is independent of specific virtualization architectures. Consequently, the user OS may be executing either in a VM (if a native VMM is used) or directly on hardware (if a hosted VMM is used). The rest of this dissertation will use the generic term *user OS* to represent either scenario. In contrast, the file system OS always executes in a VM. To reflect our prototype, figures use the native VMM architecture. Some of the features we describe are only applicable for native virtualization. For example, hosted virtualization does not support migration of the host OS.

Hence, our discussion of migration is only applicable for native virtualization.

3.3 Viability

For the FSVA architecture to be viable, four issues must be addressed. First, the FSVA interface must be stable, to avoid a repeat of the original problem of supporting changing interfaces. Second, only a few VMMs can be realistically supported by OS vendors, and the proxies should have minimal dependencies on the VMM to facilitate porting across VMMs. Third, FSVA performance must be reasonably close to native "in-kernel" file systems. Fourth, OS and VMM features must be maintained. Addressing the first two issues encourages OS vendor adoption, while solving the latter two encourages user adoption. This section discusses these issues, and describes how technology trends and appropriate design mitigate them.

3.3.1 Interface stability

For the FSVA approach to succeed, the FSVA interface (consisting of operations such as `read` and data structures such as inodes) must be stable. Otherwise, FSVAs would merely shift the location of the changing-interfaces problem: from file system developers supporting different OSs to OS vendors supporting different FSVA interfaces.

Towards that end, we designed a minimal VFS-like FSVA interface (§4.3.2). A VFS-like interface ensures interface stability because inter-OS differences tend to occur in internal OS implementation (e.g., memory management) rather than application interfaces (§2.3). The popularity of the POSIX [30] interface has led to a standard set of file system system calls that, in turn, has led to a small-ish number of VFS primitives that are common across OSs. Therefore, inter-OS differences are likely to be encapsulated in the proxies and the generic VFS-like interface ought to be unaffected.

NFS provides a successful model of a constant file system interface that has enjoyed wide OS support — though, as discussed in §2.4, it is inadequate for our purposes.

3.3.2 VMM proliferation

The user OS and FSVA proxies depend on the VMM interface. Specifically, the proxies' IPC layer depends on the VMM's shared memory and event notification interfaces. Consequently, a proliferation of VMMs could make it difficult for OS vendors to support the proxies for every VMM. Two factors mitigate this. First, there are only a few widely-used VMMs: KVM, VMware, Xen, and Microsoft dominate the marketplace [46, 47]. Second, the VMM-specific code is a small portion of the proxies — about a quarter of the code (§5.1) — and is self-contained beneath very simple interfaces.

In our experience, porting the proxies from the Xen VMM [6] to the VMware Workstation VMM [102] was relatively straightforward, especially compared to porting the FSVA proxy from Linux to NetBSD (§7).

3.3.3 Maintaining performance and the role of multicore processors

For the FSVA architecture to be practical, its performance overhead must be minimal. Although the performance-overhead sources and mitigating approaches are discussed in detail in §5.3, we now briefly describe the role of multicore processors, given their significant role in making the architecture viable.

FSVA performance overhead stems from the IPC layer. There are two components to an IPC layer: data transfer and control transfer. The overhead of data transfer is negligible when shared memory is used (§5.3). In contrast, control transfer costs dominate the FSVA performance overhead.

Multicore processors enable fast FSVA performance by decreasing the control transfer cost. Specifically, by simultaneously executing the user OS and FSVA on different cores, inter-VM control transfer is transformed to the faster operation of inter-core signaling. This avoids expensive VM and thread context switches (§5.3).

Thus, widespread multicore availability is an enabling technology trend for FSVAs. Of course, extra cores do not come for free — especially in a virtualized environment, due to server consolidation. The extra core requirement may be alleviated by gang scheduling a user VM with its FSVA(s) only during

file system-intensive periods [33, 72]. But, in general, the FSVA architecture leverages the multicore trend to enable efficient portable file systems. Others have also advocated using extra cores to enable or simplify software (e.g., for software security checks [68] or dynamic code instrumentation [19]).

3.3.4 Maintaining OS and virtualization features

Executing file systems in a separate VM requires careful design in order to maintain OS features. For example, caching is an integral aspect of OSs, often playing a critical role in application performance. Consequently, FSVAs must maintain modern unified buffer caches, among other OS features. Similarly, FSVAs must maintain virtualization features such as migration, performance isolation, and accounting. Users rely on these virtualization features. They cannot be disrupted by FSVAs.

Chapter 4 explores the consequences of the FSVA architecture on these features, and describes a practical design that maintains them.

3.4 Costs and limitations

The previous section discussed FSVA viability issues that are mitigated by technology trends or solved through appropriate design. In contrast, this section describes architectural costs and limitations that cannot be eliminated. They are intrinsic to the use of virtualization. Although we discuss these issues in relation to FSVAs, they are not specific to file systems: they arise whenever OS or application functionality is separated and executed in a virtual appliance.

3.4.1 Administration and support

FSVAs incur administration and support costs. From a user perspective, FSVAs require users to administer "extra" VMs. This includes allocating network addresses, provisioning storage, and deciding on appropriate VM resources (e.g., number of CPUs and memory size). From a developer perspective, FSVAs require file system developers to support the FSVA OS.

For example, security vulnerabilities in an FSVA require prompt developer updates.

Fortunately, administration and support costs of virtual appliances are offset by their benefits in simplifying software distribution and configuration [86, 87]. Although users must administer an extra VM, they are no longer responsible for complex software installations. Furthermore, users are free to choose any OS, without being limited by their file system's OS requirements.

Although developers are forced to provide OS support, the decoupling of the user OS and virtual appliance OS allows developers to select robust and secure OSs, such as server or embedded OSs. Furthermore, this cost is offset by the savings of not supporting diverse user OSs.

3.4.2 Overhead

The FSVA architecture adds memory and performance overhead. FSVAs introduce memory overhead due to the additional FSVA OS's memory footprint. The extra memory usage can be reduced by only including OS components required by the file system. Furthermore, because the OS executes in a VM, it does not directly access hardware and can thus avoid including most drivers, which often form a significant part of the OS memory footprint.

Sending the user OS VFS operations across an inter-VM IPC layer has an intrinsic performance overhead. VFS operations must be translated to/from a common FSVA protocol, and the IPC layer must transfer data and control between the user OS and the FSVA. Furthermore, as explained in the next chapter, FSVA design decisions necessary for maintaining unmodified file systems' semantics aggravate this overhead. Additionally, virtualization adds some overhead. Our design exploits shared memory and multicore processors to reduce the IPC layer's data and control transfer overhead, respectively.

Although performance and memory overhead can be reduced, they cannot be completely eliminated. But, they need not be eliminated. Relinquishing performance and memory resources for increased developer productivity is a recurring theme in computer science. Whether the costs are acceptable

depends on the workload and performance requirements. The evaluation chapter explores these overheads.

3.4.3 Out-of-band state

There is a fundamental limitation to the FSVA approach. By executing in a separate OS from the user OS, a file system can no longer transparently access out-of-band state such as arbitrary user OS memory or files. The user and FSVA proxies only transfer generic, file system-agnostic VFS state. As an example of out-of-band state, consider an `ioctl`[2] that includes pointers. Although the user OS proxy can transfer the opaque `ioctl` data to the FSVA, the embedded pointers would be invalid in the different FSVA address space. Another example is NFSv4 and OpenAFS' use of Kerberos authentication [67]. With Kerberos, a user runs a program to obtain credentials; the credentials are then stored in `/tmp` on a per-process-group basis. The NFSv4 VFS handlers retrieve those Kerberos credentials by accessing the `/tmp` files. With FSVAs, the file systems observe a different `/tmp` namespace, effectively hiding the user's Kerberos credentials[3].

There is no general solution to the out-of-band state problem. The use of a separate OS for the file system, an intrinsic part of the FSVA architecture, causes the out-of-band state problem. File system cooperation is necessary for transferring this state. Fortunately, most file systems do not have out-of-band state. Furthermore, for file systems with out-of-band state, the code for the extra state transfer is minimal.

3.5 Summary

The FSVA architecture leverages virtualization to enable portable file system implementations. By decoupling the file system OS from the user OS, FSVAs

[2]The Unix `ioctl` system call allows applications to send file-system-specific commands and data to a file system.

[3]At first blush, executing the Kerberos authentication program in the FSVA would solve this problem, since the credentials would be in the same `/tmp` namespace as the file system. However, this approach is not transparent to users, and would require an additional mechanism to map from the user OS's users IDs to the FSVA OS credentials.

free file system developers from having to support different OSs. A combination of technology trends and careful design make this architecture viable. Although this architecture has some performance overhead and requires additional administration, we argue that these costs are outweighed by its ability to provide portable file systems.

4 Design

This chapter describes our design goals and principles used to achieve these goals. It then details an FSVA design.

4.1 Goals

Chapter 3 described the high-level FSVA architecture, in which a file system runs in its own VM. But, this architecture permits a number of different designs, each following from a different set of design goals. This chapter describes an FSVA design intended to achieve the following goals derived from our desire to demonstrate FSVA viability (§3.3) and encourage FSVA adoption.

No file system changes To simplify adoption and deployment, file system developers should not have to modify their file system to run in an FSVA. This also accommodates legacy file system implementations.

Generality The FSVA interface should be OS- and file system-agnostic. It should not make assumptions about OS internals or file system behavior.

Minimal OS and VMM changes The user OS and FSVA proxies should require few changes to OSs. Similarly, any VMM changes should be minimal.

Maintain OS and virtualization features Applications should not be aware of the FSVA separation. Existing OS features (e.g., a unified

buffer cache, memory mapping) and virtualization features (e.g., migration, performance isolation, resource accounting) must be maintained.

Efficiency Use of FSVAs should impose minimal overheads.

These goals encourage three forms of FSVA adoption. The first two goals encourage adoption by file system developers, knowing that the OS-maintained proxies will work for them without being required to change their file system. The third goal encourages adoption by OS and VMM vendors, by simplifying the development and maintenance of the proxies. The last two goals encourage adoption by users, by maintaining performance and semantics.

There is tension among some of the preceding goals. For example, efficiency favors caching file system state in the user OS; but, doing so would require file system changes in order to maintain file systems semantics (§4.2.1), which goes against our no-file-system-changes goal. Our first priority was to encourage FSVA adoption by OS and file system vendors, through minimizing the number of OS and file system changes.

4.2 Design principles

This section discusses three major design aspects: caching, FSVA sharing, and the scope of the FSVA interface. The preceding goals lead to a corresponding FSVA design principle for each aspect.

4.2.1 Passing all VFS calls

Design principle: in order to maintain file system semantics for unmodified file systems, the user OS proxy must send all VFS operations to the FSVA.

The performance overhead of inter-VM IPCs (see §3.3.3 and §5.3) can be mitigated by caching file system state at the user OS. Data and/or metadata (e.g., inode attributes, directory entries, access control checks) can be cached. Such caching eliminates the expensive VM and/or thread control transfers (§5.3) for operations that hit the user OS cache; for example, see §6.5.

Furthermore, for data operations (i.e., ordinary and memory-mapped reads and writes), caching can eliminate shared memory *hypercalls* — synchronous software traps from a VM to the VMM. Note that this type of caching is orthogonal to the caching performed by the file system in the FSVA.

If a user OS caches file system state, then a callback scheme or cache invalidation protocol will be required. To see why, consider a distributed file system in which the `read` VFS handler checks for up-to-date data.[1] A user OS proxy that caches data would occasionally return stale results. Similarly, some file systems update a file's access time on a `read` operation. User OS caching would bypass the FSVA's file system `read` handler and, thus, break the file system semantics.

Write-back caching in the user OS can also cause problems. Many file systems carefully manage write-back policies to improve performance and achieve correctness. If the user OS performed write-back caching without giving control to the FSVA, the file system would lose this control and users would face issues such as the Memory Pressure and Write-Back issues described in Table 2.2. Such user OS proxy write-back would also break consistency protocols, like NFS, that require write-through for consistency or reliability.

To permit user OS caching while maintaining file system semantics, file systems must be modified to support a callback or cache invalidation scheme. But, file system modifications may not be desirable or feasible, especially for legacy file system implementations. Thus, although caching can improve efficiency, it hinders the ability to accommodate unmodified, arbitrary file systems. Other systems that execute a file system in its own VM, such as POFS [76], VPFS [107], and XenFS [58], have allowed caching in user OSs — but they implement a specialized virtualization-optimized file system. Our FSVA design, however, has a different goal: maintaining file system semantics for unmodified file system implementations. Instead of caching in the user OS, the FSVA design attempts to achieve efficiency through fast inter-VM IPC (§3.3.3, §5.3).

[1] NFS's 30-second staleness check is an example.

4.2.2 One user VM per FSVA

Design principle: in order to maintain OS and virtualization features with minimal changes to the OS and VMM, each user OS has a dedicated FSVA.

A fundamental FSVA design decision is whether to share an FSVA among multiple user VMs[2]. A single FSVA serving multiple user VMs provides sharing benefits. Metadata and data that are common among multiple user VMs are "automatically" shared, effectively increasing cache utilization. Indeed, this sharing capability is the primary motivation behind XenFS's single file system VM approach [58]. For stateful distributed file systems, the number of server network connections, open file handles, and cache consistency callbacks can be reduced, increasing server scalability, and greater batching opportunities exist.

There is a well-known tension between sharing and isolation [9, 53, 100], and the sharing opportunities provided by a shared FSVA design do not come for free. Implementing a unified buffer cache between a user VM and an FSVA is complicated when an FSVA is shared (§4.4.3). Additionally, a single FSVA impedes the preservation of virtualization features. For example, virtualization's effectiveness in providing performance isolation among user VMs is lost due to performance *crosstalk* [56, 9] in a shared FSVA. OS-transparent VM migration is no longer possible — an FSVA with shared file system state cannot be migrated without adversely affecting other user VMs. These difficulties are described in more detail in the following sections (see the ends of §4.4.3, §4.5.1, and §4.5.2).

As in the case for caching in the user OS, file systems and OSs can be changed to permit a shared FSVA while maintaining OS and virtualization features. But, this is undesirable given our design goal of preserving file system, OS, and virtualization features for unmodified file systems. Consequently, we have adopted a 1–to–n mapping from user VMs to FSVAs. Our experience has been that it is simpler and more fitting with our goals to involve the file system

[2]The decision to share an FSVA is only relevant when there are multiple user OSs. This occurs only for native VMMs (§3.1.1). Thus, when discussing FSVA sharing, we use the more specific term *user VM* rather than the generic *user OS* label.

with sharing, rather than involve the file system and OS with preserving a unified buffer cache and VM migration and resource accounting [1]. Note that the sharing opportunities lost by a single FSVA per user OS design are relative to a shared FSVA design, not to traditional "native" in-kernel file systems. In other words, compared to a native in-kernel file system, our FSVA design does not reduce sharing.

This design decision exacerbates the FSVA memory overhead: an extra OS is required for every user VM. Fortunately, since file system vendors are likely to only use a small subset of the OS, and they distribute a single FSVA, it is feasible for them to fine-tune the OS leading to a small OS image. Nevertheless, this FSVA design choice may not be appropriate for environments with severe memory pressure. §6.6 quantifies the memory overhead.

4.2.3 Interface scope

Design principle: in order to encourage OS vendor adoption of the user OS and FSVA proxies, each major OS type (e.g., Unix, Windows) will have its own FSVA interface.

Ideally, a single FSVA would support any and all user OSs. This would enable a file system developer to support only one OS: their FSVA's OS. Unfortunately, semantic mismatches exist between some OSs' file system interfaces. For example, Unix and Windows differ in file naming, permission semantics [105], locking granularity, and directory notifications. Consequently, an unmodified Unix file system would not provide Windows users with the full Windows file system semantics.

It may be possible to create a superset FSVA interface that supports both Windows and Unix users. Perhaps the proxies could hide the semantic differences from the file system. For example, NetApp storage appliances support both Unix and Windows clients by implementing an internal interface that is a superset of NFS and CIFS [31].

However, a "universal" FSVA interface would significantly complicate the user OS and FSVA proxies. The proxies would need to translate from/to

a complicated one-size-fits-all FSVA interface that may be different from the underlying OS's VFS interface. This is beyond this dissertation's scope.

Our design eschews a "universal" FSVA interface. Rather, we envision a single FSVA interface for every "OS type." This dissertation focuses on an FSVA interface for Unix OSs. We believe an FSVA that services most Unix OSs is possible: the popularity of POSIX [30] interface support in Unix has led to a standard set of file system system calls that, in turn, has led to a small-ish number of VFS primitives that are common across OSs. Thus, inter-OS VFS interface differences are likely to be minor, and can be encapsulated in the proxies.

4.2.4 Summary

The preceding design principles reflect our preference for minimizing FSVA adoption effort by file system and OS vendors and for maintaining OS and virtualization features. Different preferences would lead to different goals and, subsequently, different designs. Table 4.1 contrasts the FSVA design with related systems.

We preferred minimizing file system changes over performance. If file system vendors have an opposite preference, caching in user OSs may be more appropriate, leading to higher performance. Furthermore, maintaining virtualization features was an important goal for us. But, in some environments, such as high performance computing (HPC), VM migration and performance isolation are not required. A shared FSVA would be more appropriate in those environments. Similarly, specialized HPC environments may not require maintaining OS features such as a unified buffer cache or memory mapping; this would lead to simpler proxies.

For the remainder of this dissertation, the term "FSVA" will refer to the design point specified by the preceding design principles.

4.3 Design overview

Our FSVA design is based on the architecture described in §3.2. A file system executes in its own VM that runs the preferred OS of the file system

System	User OS caching	Shares FS VM	File system scope	Maintains FS semantics	Maintains UBC	Maintains VMM features
POFS [76]	Yes	Yes	Single FS	Yes	No	No
XenFS [58]	Yes	Yes	Single FS	Yes	No	No
VNFS [58]	Yes	Yes	NFS	Yes	No	Yes (with minor changes)
VPFS [107]	Yes	No	Arbitrary FS	No	No	n/a (microkernel)
VMware Workstation Shared Folders [101]	Yes	n/a (hosted VMM)	Arbitrary FS	No	No	n/a (hosted VMM)
FSVA	No	No	Arbitrary FS	Yes	Yes	Yes

Table 4.1. Comparing design decisions and capabilities of file system (FS) VMs. The FSVA design is the only one that maintains a unified buffer cache, virtualization features, and file system semantics for arbitrary file systems. POFS [76] and XenFS [58] implement a specialized virtualization-optimized file system; they do not support arbitrary file systems. VNFS [58] optimizes NFS access for NFS clients physically colocated with a server; the stateless NFS nature allows the VNFS layer to simply recreate all NFS connections on a migration, thus preserving that virtualization feature. Resource accounting and performance isolation are not affected by VNFS any more than if the NFS operations went through the local networking layer. VMware Workstation Shared Folders [101] and VPFS [107] are the only other systems that support arbitrary file systems, but both systems perform caching in the user OS; this fails to preserve the file system semantics.

developers. Users run their applications in a separate OS, possibly in a VM, using their preferred OS. File system-agnostic proxies in both OSs efficiently pass VFS operations from the user OS to the FSVA and maintain OS and virtualization features. Furthermore, our FSVA design is based on the three design principles described in the preceding section: all user OS VFS operations are sent to the FSVA (i.e., there is no caching in the user OS), FSVAs are not shared among user OSs, and there is an FSVA for each major OS type (e.g., Unix, Windows).

The user OS proxy registers as a file system in the user OS. Application system calls or background VFS operations (e.g., writeback) invoke the proxy's VFS handlers. The majority of the proxy's VFS handlers are simple: they encode the VFS operation and its arguments into an OS-independent format, perform an IPC to the FSVA, wait for a response, decode the response, and reply to the user OS with the FSVA's result. On receiving a request, the FSVA proxy decodes it, sends it to the file system, encodes and sends the file system's response to the user OS proxy. As discussed in §4.2.1, all user OS VFS operations are sent to the FSVA in order to preserve file system semantics. By synchronously waiting for the FSVA response, just like the caller would do if the file system was local, the user OS proxy maintains file system behavior.

4.3.1 IPC layer

Our IPC[3] layer closely mirrors the Xen [6] block and network drivers' IPC layers. The IPC layer performs two tasks: data transfer and control notification. Data is transferred using an asynchronous ring buffer residing in a shared memory region. The ring buffer, split into equally-sized slots, contains two producer-consumer queues: one for requests and one for responses. The two queues consist of equally-sized slots. The user OS proxy enqueues operations to the request queue, which are then dequeued by the FSVA proxy. Responses are handled in a reverse manner: the FSVA proxy enqueues responses to the

[3]Strictly speaking, an IPC layer provides communication between processes, not VMs. But, both cases contain similar protection boundary crossings and thus share a similar structure. Consequently, we use the traditional *IPC* label.

response queue, which are then dequeued by the user OS proxy. Figure 4.1 illustrates the IPC ring. Multiple outstanding VFS operations at the user OS can lead to multiple requests in the IPC ring. To avoid deadlocks like those described in Table 2.2, the proxies do not perform memory allocations in the IPC path.

The Xen IPC layers use inter-VM software interrupts to signal pending requests and responses. To achieve low IPC latency, our IPC layer also supports using dedicated CPU cores in order to avoid VM and thread switches. Details are in §5.3.

There are two control advantages to the asynchronous ring buffer structure. First, the single-reader single-writer structure enables non-blocking synchronization between the user OS and the FSVA [66]. Writers limit the number of enqueued requests/responses, ensuring that the two queues occupy distinct slots on the ring. Second, it allows out-of-order responses from the FSVA. This flexibility permits low latency when multiple outstanding requests have significantly different latencies (e.g., an in-cache `getattr` versus an out-of-cache `read`).

The majority of IPCs are triggered by the user OS. However, because FSVA pages that are memory-mapped in the user OS may be invalidated in the FSVA (e.g., due to cache consistency callbacks or FSVA file system applications), the FSVA proxy may need to trigger a synchronous IPC to the user OS. Consequently, our IPC layer consists of two rings: one ring allows IPCs from the user OS to the FSVA, and the other ring allows IPCs in the reverse direction.

4.3.2 FSVA interface

The FSVA interface consists of the common Unix VFS operations, based on an examination of the Linux, NetBSD, and OpenSolaris VFS interfaces. Most Unix OSs have similar VFS interfaces, both in the operation types (e.g., `open, create, write`) and state (e.g., file descriptors, inodes and directory entries). Consequently, the Unix VFS interfaces are similar and differences can be normalized by the proxies. In addition to the VFS operations, the

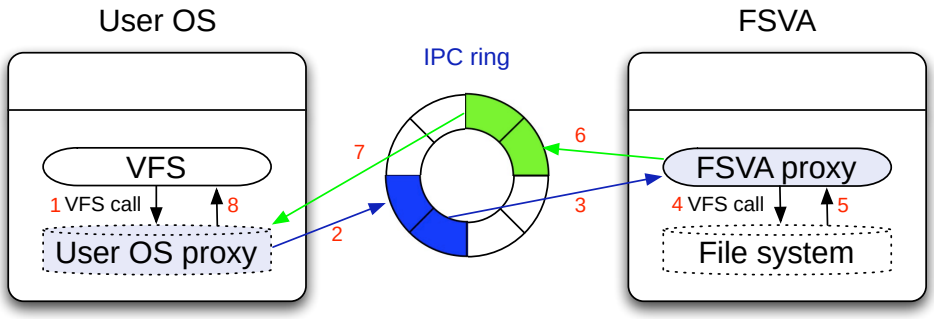


Figure 4.1. This figure illustrates the sequence of operations and data movement for handling a user OS VFS operation. The red numerals indicate the sequence. Note that data operations (e.g., `read`) require more steps in order to share memory between VMs.

FSVA interface includes calls to support a unified buffer cache and migration. Table 5.2 lists the FSVA interface.

What has been left out of the FSVA interface is notable: virtual memory interactions, data and metadata caching, device access, memory allocation, locking, preemption policy, and threading. It is precisely these aspects that change most across OSs (versions) and cause the most grief for file system developers (§2.3). The spartan FSVA interface supports the hope that it can remain constant among OSs and across OS revisions. The FSVA interface does not constrain the functionality of the user OSs or the file system. OS developers are free to change internal OS interfaces and implementation, as long as they maintain the proxies.

4.3.3 Data operations

To avoid unnecessary data copies, VFS data operations require special handling in the user and FSVA proxies. Modern OSs provide multiple file I/O interfaces: ordinary `read` and `write` calls, direct I/O, asynchronous I/O, and memory-mapped I/O. We describe the use of shared memory for ordinary I/O; the other I/O interfaces have similar implementations.

For ordinary `read` and `write` operations, an application provides the OS with a buffer that will receive or provide the data, respectively. To preserve

file system semantics, every `read` and `write` call is handled in the FSVA, bypassing the user OS's generic `read/write` handlers and buffer cache. In order to avoid memory copies, the application buffer is mapped into the FSVA's address space. The user proxy uses the VMM's shared memory facility to grant the FSVA access to the application buffer pages. The FSVA proxy then maps those pages into its address space. Thus, the file system implementation transparently accesses the application buffer.

4.4 Maintaining OS features

This section describes how user OS features are maintained, despite the file system executing in a separate VM.

4.4.1 Metadata duplication

Many OS components, outside the VFS layer, directly access generic file system metadata, such as inodes or directory entries. For example, an OS's program-loading module interacts with programs' open file descriptors and inodes. To preserve the ability to execute programs stored in an FSVA file system, the user OS proxy creates this metadata in the user OS. The FSVA will also contain metadata, to support the FSVA OS and file system. Thus, metadata exists in both the user OS and the FSVA OS. Note that the user OS metadata is minimal: the user OS proxy creates basic inodes and directory entries, but any file system-specific "extra" metadata (e.g., block allocation maps) is stored only in the FSVA. This follows from the file system-agnostic FSVA design — the proxies are not aware of file system-specific metadata.

The user OS's caching is a read- and write-through *functionality-supporting* cache. Although read-only and mutating VFS operations are always handled in the FSVA, the cache enables existing user OS features such as memory mapping and file system-based process execution that directly access file system metadata to continue working.

Metadata duplication can be avoided through invasive OS changes that wrap all metadata access in the user OS and perform corresponding IPCs. From a software engineering standpoint, this is how it should work. But,

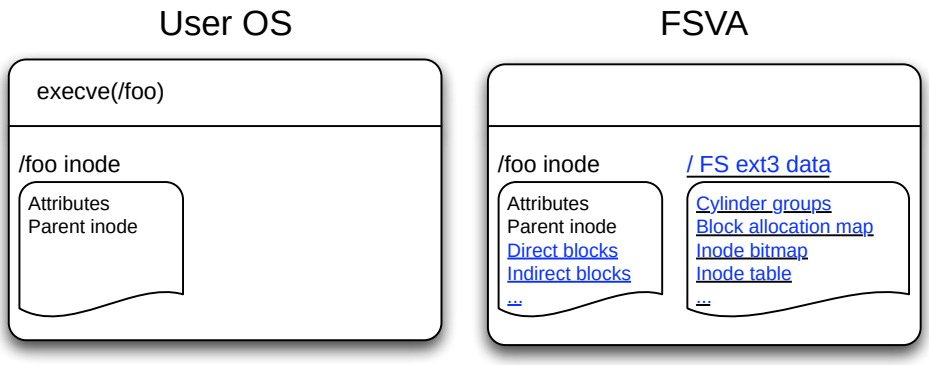


Figure 4.2. Metadata is duplicated between the user OS and FSVA OS, to maintain user OS functionality. The user OS proxy creates generic metadata structures in the user OS. File system-specific metadata, shown underlined for a Linux ext3 file system example, is only kept in the FSVA OS.

practically, this would complicate the adoption of the user OS proxy by OS vendors. Given that inodes and directory entries are small data structures, we opted for duplication. As described in §4.4.3, data blocks are not duplicated.

For distributed file systems with cache consistency callbacks, a user OS might contain stale metadata. For example, a cache consistency callback can update the FSVA's file's attributes. But, this inconsistency will not be visible to user applications. OSs already call into the file system in response to application operations that require up-to-date metadata. This will cause the user OS proxy to perform an IPC to the FSVA in order to retrieve up-to-date metadata.

4.4.2 Security and other common VFS features

The VFS layer contains file system-agnostic features, such as permission checks, security auditing, and directory notifications. It also performs some locking on behalf of the file system VFS handlers. In contrast with the VFS layer interface, these features vary widely across OSs.

Maintaining the user OS's security checks and policies is required in order to preserve application semantics. Most Unix OSs perform access control in the VFS layer, above the file system. Since the user OS proxy sits below

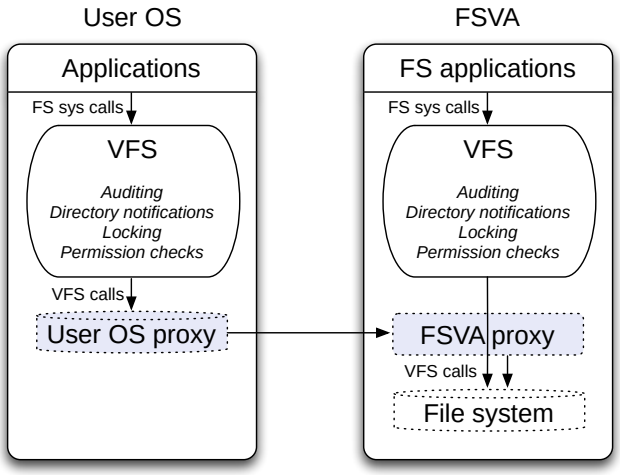


Figure 4.3. The VFS layer contains file system-agnostic features such as permission checks, auditing and directory notifications and performs some locking on behalf of the VFS handlers. These features continue to function with FSVAs, because the user OS proxy is underneath the VFS layer stack. The FSVA proxy avoids duplicating this functionality by directly calling the file system's VFS handlers.

the VFS layer, the existing VFS security checks continue to work. In the FSVA, the proxy directly calls into the file system, thereby bypassing the FSVA OS's security checks. In contrast to generic OS security checks, file system-specific security features, such as Kerberos authentication [67], may require extra effort, as discussed in §3.4.3.

Other common VFS features are similarly unaffected by FSVAs, because the user OS proxy is underneath the VFS layer stack.

In different FSVA designs, executing a file system in a VM provides security properties by enforcing a narrow interface from/to the file system (e.g., [107, 59]). This dissertation does not explore this avenue: our FSVA design relies on mutual trust between the user VM and FSVA. For example, the user VM is trusted by the FSVA to correctly identify users and to store security credentials such as Kerberos tickets. Similarly, the FSVA is trusted by the user OS to return the appropriate file system responses.

4.4.3 Unified buffer cache

Managing a computer's limited physical memory is an essential OS function. In particular, the majority of memory is typically used by two OS components[4]: virtual memory and file systems. The virtual memory cache consists of application code and data pages (i.e., program stack and heap). The file system caches largely consist of data blocks (the *buffer cache*), as well as metadata. Because metadata is duplicated between the two OSs (§4.4.1), the file system cache we are concerned with is the buffer cache.

Early Unix OSs had separate virtual memory and file system caches. This had data and control disadvantages. First, disk blocks were duplicated in both caches. Second, the lack of a single eviction policy led to suboptimal cache partitioning. Unified buffer caches (UBCs) (also known as page caches) fix both problems [37, 91]. A single cache stores both virtual memory pages and file system data, avoiding copies and enabling a single eviction policy.

An analogous problem exists for FSVAs: separate caching between the user OS and FSVA OS. User applications run in the user OS; therefore, the virtual memory cache exists in the user OS. The file system runs in the FSVA; therefore, the buffer cache exists in the FSVA. Without extensive OS changes, we cannot coalesce the two OSs' caches into a single cache — the OSs have different data structures and expect exclusive access to hardware (e.g., per-page access and dirty bits). Instead, we maintain the illusion of an inter-VM unified buffer cache by using shared memory (to avoid data copies) and by loosely coupling the two caches (to obtain a single eviction policy). The user OS and FSVA proxies maintain this illusion with minor changes to the FSVA OS and no changes to the user OS.

FSVAs avoid unnecessary data duplication by using the VMM's shared memory facilities. There are two types of data operations to consider: ordinary and memory-mapped. Ordinary `read` and `write` operations are always sent to the FSVA (§4.3.3). Because the user OS is not involved in executing these

[4] In memory-constrained systems, such as embedded systems or high performance computing, paging is unavailable. Virtual memory is carefully managed and pinned in physical memory. Consequently, these systems lack a virtual memory cache. This section assumes paging is enabled and hence a virtual memory cache exists.

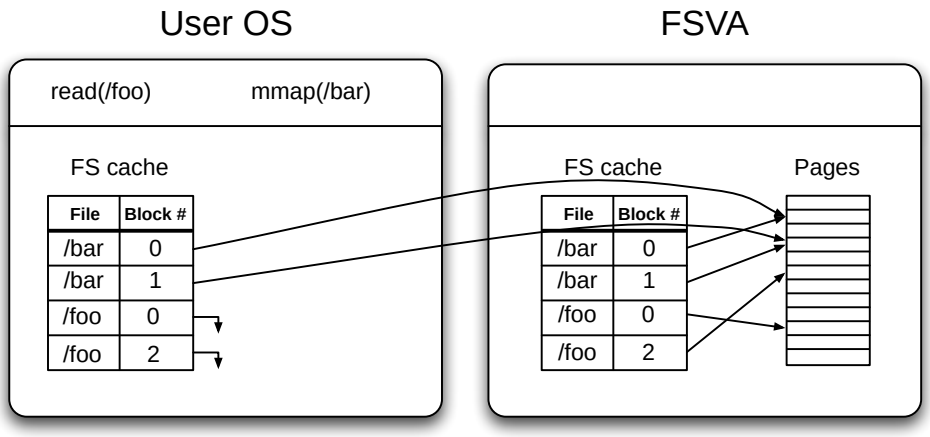


Figure 4.4. A unified buffer cache between the user OS and FSVA avoids data block duplication and enables a single eviction policy. To support memory-mapped I/O, the user OS proxy maps the relevant FSVA pages into the user OS address space. Ordinary data operations do not require similar page mappings, because the operations are always executed in the FSVA. Ghost entries in the user OS (shown with a null pointer) are used to track the corresponding FSVA page's access frequency without an associated page mapping.

operations, the user OS caches do not contain the corresponding data blocks. Therefore, ordinary data operations do not create duplicate data blocks in the user OS and FSVA. Inter-VM shared memory enables the FSVA file system to directly copy data from/to the application buffer in the user OS. In contrast, memory-mapped I/O and program execution[5] require the presence of data blocks in the user OS's cache — the page fault handling code that lazily reads in data blocks requires the relevant data blocks to be in the user OS cache. To avoid duplication of data blocks, the user OS proxy maps the FSVA pages that contain the data blocks into the user OS's address space. Figure 4.4 illustrates the two situations.

Providing a single eviction policy enables optimal sizing of the two VMs' memory, based on the combination of the virtual memory workload in

[5]Most Unix OSs lazily page-in a program's code pages. From a caching perspective, this is effectively memory-mapped I/O.

the user OS and the buffer cache workload in the FSVA. However, this is complicated because each OS observes different types of memory allocation and accesses. On one hand, because applications execute in the user OS, the user OS allocates virtual memory pages and observes their access behavior. On the other hand, since I/O is performed in the FSVA (both in response to user requests and due to latent file system activities such as readahead and writeback), the FSVA allocates file system buffer pages and observes their access behavior.

FSVAs bridge this semantic gap by informing one OS of the other OS's memory allocations and accesses. To support multiple FSVAs and to preserve the user OS's cache eviction policy, we chose to inform the user OS of the FSVA's file system memory allocations and accesses. Thus, the user OS controls the eviction policy.

The FSVA proxy registers callbacks with the FSVA buffer cache's allocation and access routines. When the FSVA proxy observes that a new page is inserted into the buffer cache, it makes a hypercall to grow the FSVA memory size by a single page[6]. On every IPC response to the user OS, the FSVA proxy piggybacks page allocation and access information. On receiving a page allocation message, the user OS proxy makes a hypercall to shrink the user VM, thereby balancing the memory usage among the VMs. In addition, the user OS proxy adds a *ghost page* [29, 64, 75] entry to its cache. A ghost page is a cache entry that lacks a physical backing page. Ghost pages are used as a placeholder in the user OS to track the corresponding FSVA buffer page's access frequency. Figure 4.4 illustrates ghost entries.

On receiving a page access message, the user OS proxy updates the ghost page's recency in the user OS's cache. When memory pressure later causes the user OS to evict this ghost page, the user OS proxy grows the user VM by a page and the FSVA proxy shrinks the FSVA by a page. The net result is a coupling of the two OSs' unified buffer caches and a single inter-VM cache eviction policy, controlled by the user OS.

[6]To support memory ballooning [103], VMMs provide hypercalls that grow and shrink a VM's memory size.

Ballooning [103] is orthogonal to our inter-VM unified buffer cache technique. It is primarily a *mechanism* for changing a VM's memory size; the VMM contains heuristics for a global optimization of VMs' memory sizes. In contrast, our inter-VM unified buffer cache maintains the existing user OS cache eviction *policy* while coupling the user VM and the FSVA.

Our design choice of a single FSVA per user VM (§4.2.2) greatly simplifies the inter-VM unified buffer cache design. In a shared FSVA design, concurrent user requests and latent file system operations (e.g., readahead and writeback) complicate proper attribution of page allocations and accesses to the corresponding user. The FSVA OS and the file system would require many modifications to ensure proper attribution.

4.5 Maintaining virtualization features

The FSVA design preserves virtualization features. Our design decision of not sharing an FSVA among multiple user VMs maintains inter-VM performance isolation and resource accounting. Although VM migration also benefits from this design decision, additional work in the proxies is required.

4.5.1 Performance isolation and resource accounting

Virtualization provides coarse-grained physical resource sharing among users. This enables inter-VM performance isolation and accurate resource accounting. These features are important in utility computing.

Sharing an FSVA among multiple user VMs would disrupt performance isolation among VMs. For example, heavy activity (e.g., opening a large number of files) by one user VM would affect the performance of other user VMs. Modifying the FSVA OS and file system to avoid performance crosstalk is quite challenging [56, 9]. Our single user VM per FSVA design avoids introducing these problems at this level.

FSVA resource usage should be charged to the user VM, to continue allowing an administrator to set a single coarse-grained resource policy for user VMs. Logically, the user VM and its FSVAs form a single unit for the purpose of resource accounting.

Associating only a single user VM per FSVA simplifies resource accounting. If multiple VMS share an FSVA, the VMM would not be able to map FSVA resource utilization to user VMs. Instead, the FSVA would itself have to track per-user resource usage and inform the VMM. For shared block or network driver VMs [34], tracking per-user resource usage is viable, owing to the small number of requests types and their fairly regular costs [40]. But, FSVAs provide a much richer interface to users: there are many VFS operation types, and an operation can have significantly varying performance costs (e.g., cache hits versus misses). Latent OS work (e.g., writeback) further complicates OS-level resource accounting. Thus, our design of one user VM per FSVA simplifies resource accounting by leveraging the VMM's existing coarse accounting mechanisms.

4.5.2 Migration

Migration is an important virtualization feature, providing high availability during scheduled downtime and enabling load balancing. VMMs support migration of unmodified OSs. Furthermore, *live migration* minimizes VM downtime through background state copying before a VM is suspended and restored [20].

If a migrating VM relies on another VM for a driver, the migrating VM's driver connection is reestablished to the driver VM in the new physical host [34]. This is relatively simple since driver VMs are (mostly) stateless and provide idempotent operations.

FSVAs complicate migration. In contrast to driver VMs, FSVAs potentially contain state on behalf of a user VM, and the FSVA interface is non-idempotent. There are three possible approaches to dealing with migration [28]: either the user VM and its FSVA could be simultaneously migrated, the user VM could continue communicating with the FSVA at the original physical host, or a new FSVA could be created at the new host and populated with migrated file system state. Our use of shared memory and desire to maintain memory mapping support preclude the second option, as conventional networking cannot efficiently support these features. The

third option is also undesirable, because it requires file system cooperation in migrating internal state (such as extra metadata or cache consistency callbacks for distributed file systems) to the new FSVA; this would break our design goal of no file system changes.

To support migration for unmodified file systems, we migrate an FSVA along with its user VM. This approach exploits VM migration's existing ability to transparently move VMs. Because some file system operations are non-idempotent, care must be taken to preserve exactly-once semantics (§5.6). Another complication is that shared memory pages (e.g., for the IPC ring and memory-mapped I/O) will likely have different physical page mappings after migration. To address these two issues, the user OS and FSVA proxies transparently restore the shared memory mappings and retransmit any pending requests and responses that were lost during the IPC layer teardown. Moreover, we retain live migration's low downtime by synchronizing the two VMs' background transfer and suspend/resume phases.

Having only a single user OS per FSVA simplifies migration. In contrast, a shared FSVA would require file system involvement in migrating private state belonging to the specific user OS being migrated. Additionally, for stateful distributed file systems, the server would need to support a client changing its network address. This would likely require server modifications. By migrating the unshared FSVA, our approach exploits the existing migration feature of retaining network addresses, thereby avoiding server changes.

5 Implementation

To demonstrate the feasibility of and to evaluate the FSVA architecture and design, we implemented an FSVA prototype. This chapter details the implementation of this prototype.

5.1 Prototype overview

We implemented an FSVA prototype using the Xen [6] virtualization platform. Xen was chosen because of its efficiency and source code availability. User VMs and FSVAs execute as unprivileged VMs. To demonstrate intra-OS file system portability, we implemented the user OS and FSVA proxies for two Linux kernel versions: 2.6.18 (released in September 2006) and 2.6.28 (released in December 2008). To demonstrate inter-OS file system portability, we also implemented the FSVA proxy for NetBSD 5.99.5 — that port currently is full-featured except for migration support. Linux and NetBSD were chosen because of their mature Xen support.

The user OS and FSVA proxies are largely implemented as self-contained kernel modules. But, several changes to the Linux and NetBSD kernels were necessary to support memory-mapping (§5.4) and a unified buffer cache (§5.5). In total, the Linux changes constituted less than 100 source lines of code (SLOC), as measured by SLOCCount [24]. The majority of these changes enable a user OS to memory-map FSVA pages. Consequently, because our NetBSD implementation only supports an FSVA proxy, the core NetBSD kernel changes were only 4 SLOC. The small size of the Linux and NetBSD kernel changes reflects the efficacy of the FSVA design in maintaining file system and OS features with minimal OS changes.

Xen was modified in four ways. First, 295 SLOC were added to "domain 0"'s[1] user-level migration code to enable atomic migration of two VMs while preserving the low downtime of live migration (§5.6). Second, 42 SLOC were added to the core Xen kernel to implement our new IPC notification mechanism (§5.3). Third, 91 SLOC were added to user-level scripts to enable system administrators in "domain 0" to create connections between user VMs and FSVAs. Fourth, a single-line change was necessary to support memory-mapping after migration (§5.4). Note that the first two modifications are not FSVA-specific: they permit atomic migration of any VM pair and provide a new type of inter-VM control notification, respectively.

The Linux user OS and FSVA proxies contain ∼4700 and ∼3500 SLOC, respectively. Of the sum, ∼900 belong to the unified buffer cache code and ∼400 support migration. Thus, the majority of the proxies' code supports the core operation of transferring and handling VFS operations. Supporting the unified buffer cache and migration does not overly complicate the implementation. The NetBSD FSVA proxy contains ∼3100 SLOC — it is smaller than the Linux FSVA proxy because it currently lacks migration support. Table 5.1 lists the FSVA code breakdown. As a reference point, the Linux NFSv3 client code is ∼13,000 SLOC.

5.2 FSVA interface

The majority of VFS operations have a simple implementation structure. The user OS proxy's VFS handler finds a free slot on the IPC ring, encodes the operation and its arguments in a generic format, and signals the FSVA of a pending request via an event notification (§5.3.2). Upon receiving the notification, the FSVA proxy decodes the request and calls the file system's corresponding VFS handler. Responses are handled in a reverse fashion.

Table 5.2 lists the FSVA interface's operations. Most of the IPCs correspond to VFS calls such as `mount`, `getattr`, and `read`. As described below,

[1] "Domain 0" is a privileged Xen VM. It is the only VM in which administrative operations such as creation and deletion of VMs can be executed.

	Core	IPC	UBC	Migration	**Total**
Interface definition		512			512
Linux user OS					
Kernel module	2701	1372	399	202	4674
OS changes	89				89
Linux FSVA					
Kernel module	2011	881	472	163	3527
OS changes			10		10
NetBSD FSVA					
Kernel module	2009	504	546		3059
OS changes			4		4
Xen					
Core VMM		42		1	43
User-level		91		295	386
Total	6810	3402	1431	661	12304

Table 5.1. Per-component source lines of code counts for various FSVA functionality. Blank entries indicate that a component did not include any code for that functionality. *Core* refers to the bulk of the FSVA code that intercepts and encodes VFS operations (at the user OS) and decodes and implements them (at the FSVA). *Core* also includes memory-mapping support.

there is also an IPC to support migration and two IPCs to support a unified buffer cache.

File identification was a surprisingly challenging aspect of the implementation. Identifying files using their full pathname on every IPC is inefficient due to the extra path lookups. Initially, we used inode numbers as a unique file identifier and relied on internal Linux kernel functions for translating an inode number to an inode. But, the Linux kernel removed this mapping function after the Linux 2.6.18 kernel, posing problems for our Linux 2.6.28 kernel port. We then attempted to use the existing VFS handlers that encode/decode an opaque file handle; they are provided to support exporting a file system over NFS. However, only a minority of file systems support this optional interface. Finally, we used FSVA pointers to in-memory

Type	Operations
Mount	`mount, unmount`

Metadata	`getattr, setattr, create, lookup, dentry_validate`
	`mkdir, rmdir, link, unlink, rename, truncate`
	`readdir, symlink, readlink, dirty_inode, write_inode`

File ops	`open, release, seek`
Data	`read, write, map_page, unmap_page`
Misc.	`flush, fsync, permission`
File ID	`dentry_release`
UBC	`invalidate_page, evict_page`
Migration	`restore_grants`

Table 5.2. FSVA interface. Most of the calls correspond to VFS operations, with the exception of three IPCs that support migration and a unified buffer cache.

directory entries as file identifiers. This mechanism works for all file systems but requires careful reference counting.

When an FSVA proxy handles a `lookup` operation, the resulting directory entry's reference count is incremented. (We use directory entries, not inodes, because directory entries disambiguate between hard links.) The user OS proxy treats the FSVA directory entry pointer as an opaque identifier. Incrementing the FSVA directory entry's reference count ensures that the directory entry is not garbage collected while the user OS proxy holds a pointer to the directory entry. When the user OS proxy drops its reference to the directory entry (either because of metadata cache pressure or because the file is deleted), the user OS proxy informs the FSVA proxy that it is safe to decrement the corresponding reference count. These notifications are usually piggybacked on IPC requests. If there are too many queued notifications, the user OS proxy sends a synchronous `dentry_release` IPC.

To enable inter-operability between 32- and 64-bit OSs, we use compiler directives to ensure 32-bit IPC structure alignment. There is no need for XDR functions [94] as the two VMs share the same endianness. Due to

idiosyncrasies in 32-bit Linux's "high memory" implementation [5] and Xen's shared memory facility, we can only map pages between VMs that are in the low memory region. This places a limit on the number of shared pages between VMs if one of them is 32-bit. We believe Xen can be modified to remove this limitation, but we have not done so.

5.3 IPC layer

To maintain isolation between VMs, Xen does not allow arbitrary VMs to communicate with each other. A system administrator must first allow inter-VM communication in "domain 0". Consequently, we modified the "domain 0" management console scripts to support installing and removing connections between a user VM and an FSVA. When a connection is initiated, the user OS and FSVA proxies set up an IPC layer: a shared memory region (containing the I/O ring of requests and responses) and an event notification channel (for inter-VM signaling). This IPC layer closely resembles Xen's block and network drivers' IPC layers [6].

5.3.1 Data transfer

There are two types of application I/O: ordinary `read`/`write` and memory-mapped `read`/`write`. For ordinary I/O, the application provides a user-space buffer. The user OS proxy creates a sequence of *grants* for each page in the application buffer — each grant covers only one page — using Xen's shared memory facility. No hypercalls are involved in this operation: the user OS proxy writes its intention in a shared memory region with Xen. The grants are then passed in the IPC request structure. In turn, the FSVA proxy maps the grants into the FSVA address space using a single hypercall, calls the file system to perform the I/O directly to/from the buffer, unmaps the grants using a hypercall, and sends the I/O response to the user. The user OS proxy can then destroy the grants. As an optimization, if less than 4 KB of data is read or written, data is copied back and forth using *trampoline* buffers — pages that are shared during proxy initialization — because the cost of the

shared memory hypercalls is not amortized over the small access size (see §6.4).

Memory mapped I/O is handled in a similar fashion, except that the roles of grant issuer and user are reversed. When an application memory access causes the OS page fault handler to request a data page from the file system, the user OS proxy performs a `map_page` IPC to the FSVA. In response, the FSVA proxy calls the file system to bring the relevant page into the FSVA buffer cache, pins the page in memory, and returns a grant for the page in the IPC response. The user OS proxy then maps that grant into its buffer cache using a hypercall. Once the user OS unmaps the page, the user OS proxy queues an FSVA notification to unmap the page. These notifications are piggybacked on future IPC requests. If there are too many queued notifications, the user OS proxy performs a synchronous `unmap_page` IPC. Note that, because the file system in the FSVA allocates the data page, this reversal of grant issuer/user roles is necessary.

5.3.2 Control notification

Our design goal of supporting unmodified file systems does not come for free. It forces all VFS calls to be sent to the FSVA, thereby increasing inter-VM IPC frequency. In turn, FSVA performance is highly dependent on the IPC layer's performance. We now discuss the overheads associated with the traditional Xen IPC mechanism and describe a mitigating technique.

There are two components to an IPC: data transfer and control transfer. Data transfer is fast (less than 1 μs) because requests and responses are small[2] and are stored in a shared memory region. Control transfer has two elements: 1) inter-VM signaling and 2) VM- and OS-level scheduling and context switching. If the user VM and FSVA are concurrently executing on different cores, then VM-level scheduling and context switching can be avoided. But, inter-VM signaling must still be performed as well as OS-level scheduling and context switching, as we now describe.

[2]Requests and responses are 512 bytes, including piggybacked notifications.

The standard Xen mechanism for inter-VM signaling employs *event channels* [6]. During IPC layer creation, two VMs create an event channel and transfer the event channel identifier in an out-of-band channel. Subsequently, when a VM wishes to notify another VM of an IPC request/response, it performs a "send event" hypercall. This hypercall sends a software interrupt to the destination VM. If the VM is not currently executing or has masked that interrupt, the interrupt is marked in its pending interrupts mask. Otherwise, an inter-processor interrupt (IPI) is sent to the CPU executing the other VM.

Upon receiving an IPI, the CPU invokes the OS's interrupt handler. This causes a thread context switch: the current processor state must be saved before executing the interrupt handler thread. In most Unix OSs, the interrupt handler typically masks off other interrupts and cannot sleep. Thus, the interrupt handler does not execute general-purpose kernel code that may block. So, the event channel interrupt handler signals a worker thread that then actually handles the operation. This requires a second thread context switch. On our servers running Linux 64-bit x86, a thread context switch costs $\sim$3.5 μs (§6.4). Thus, a *one-way* inter-VM signal costs 7 μs in thread switch times. Inter-VM signaling adds an additional $\sim$2 μs each way — for the "send event" hypercall and its corresponding IPI.

The Xen event channel mechanism was designed for I/O devices, in which a two-way IPC signaling overhead of 18 μs is insignificant compared to hardware access latencies. But, this overhead is too high for FSVAs: many VFS operations (e.g., `getattr`, `permission`) execute in less than 1 μs.

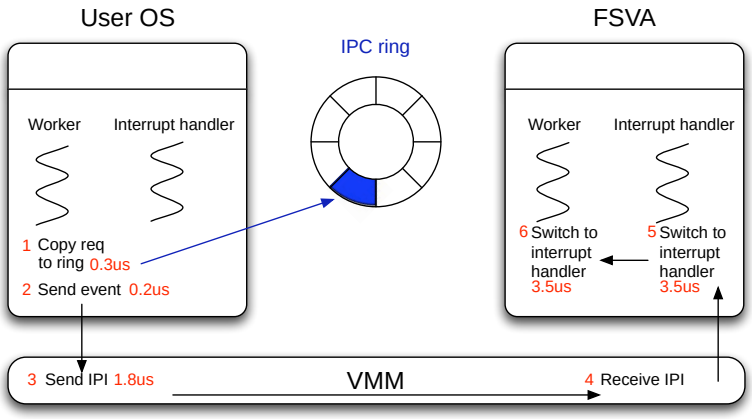
When multiple processors are available, a well-known technique for reducing IPC cost is to use polling as a signaling mechanism [7]. We implemented a polling version of our IPC layer, in which a worker thread busy-spins on a shared memory location waiting for notification of a request/response notification. This avoids both thread context switches as well as the inter-VM signaling cost (i.e., the "send event" hypercall and IPI). As a result, the null IPC latency drops from 21 μs to 4 μs (§6.4). But, polling is energy-inefficient during idle periods: the CPU core is continuously monitoring a shared memory address.

Fortunately, x86 processors include instructions that provide polling-like latency with events-like energy-efficiency. These instructions were introduced to avoid this type of polling for inter-process synchronization. The `monitor` and `mwait` instructions put a processor core in low-energy mode until a write occurs to a specific memory address. We used these instructions to implement a new inter-VM notification mechanism. Because these are privileged instructions, we added a new hypercall to wrap these instructions. Our `mwait`-based IPC has similar latency to the polling IPC (§6.4), with a slight increase due to the cost of a hypercall. Figure 5.1 illustrates the events- and `mwait`-based IPC mechanisms.

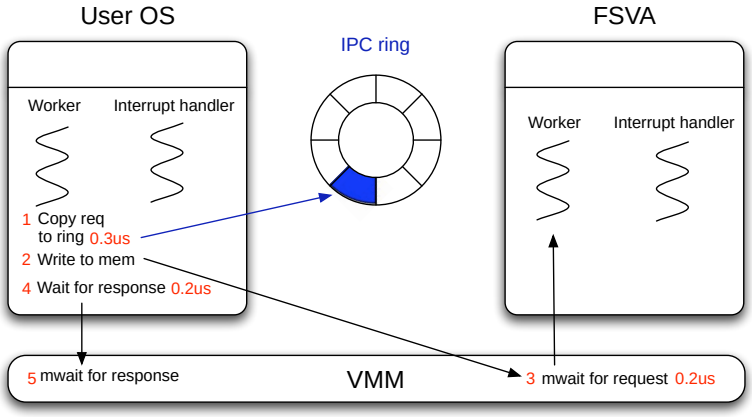
5.4 Memory mapping

Memory-mapped I/O avoids the memory copies associated with ordinary `read`/`write` calls. To allow user OS applications to memory-map FSVA pages (thus avoiding data duplication), we had to add several hooks to the Linux page fault handling code. These hooks were required because Xen would normally block the user OS's attempt at setting a page table pointer to point to another VM's pages. Our generic hooks enable a file system (in this case, the user OS proxy) to execute the Xen hypercall for setting/clearing user-space page table entries that point to another VM's pages.

When the user OS memory-maps FSVA pages, care must be taken to ensure that the FSVA does not invalidate those pages. The pages will not be invalidated through eviction, because the FSVA proxy pins the page in memory during the memory map IPC. Thus, normally, the user OS will unmap the shared page before it can be invalidated in the FSVA (say, due to eviction or file deletion). However, in some rare cases, the FSVA may trigger a page invalidation. For example, a cache consistency callback might invalidate a page. To prevent a segmentation fault in the user OS, we added a hook to the page invalidation routine in the FSVA OS. If an invalidated page is memory-mapped in the user OS, the FSVA proxy synchronously requests the user OS to unmap the page. To simplify our implementation,



(a) IPC control notification using Xen event channels.



(b) IPC control notification using our new `mwait` hypercall.

Figure 5.1. One-way control path and latencies for two different IPC notification mechanisms: Xen even channels and our new `mwait` hypercall. Event channels require an inter-processor interrupt as well as two thread switches, for each direction in the IPC. In contrast, the `mwait`-based IPC avoids these operations: when the source VM writes to a specific shared memory location, the destination VM immediately returns from the Xen `mwait` hypercall into the worker thread.

we avoided placing these "reverse" IPCs on the existing IPC layer. Instead, a separate "reverse IPC layer" supports IPCs that originate in the FSVA.

5.5 Unified buffer cache

To maintain a unified buffer cache, the user OS proxy must be notified of FSVA buffer cache allocations and accesses. We added hooks to Linux and NetBSD to inform the FSVA proxy of these events. When either event occurs, the FSVA proxy queues a notification. The notifications are piggybacked on IPC responses. But, there is a limit to how many notifications can be piggybacked on each fixed-sized IPC response. Therefore, during periods of high FSVA buffer cache activity, the user OS proxy performs synchronous IPCs to drain the queued unified buffer cache notifications.

Linux allocates buffer cache pages in only one function, simplifying the capture of page allocation events. For page access events, there are two ways in which a page is marked as accessed. First, when a file system or the generic VFS code looks up a page in the buffer cache, the search function automatically marks the page as accessed by placing it on an "active pages" list. We added a hook to this mechanism. Second, the hardware memory controller sets the page accessed bit for page table entries when their corresponding page is accessed. This mechanism is necessary for virtual memory pages in order to capture page-access information for arbitrary pages that get accessed outside the buffer cache. Periodically, Linux scans the page table entries to update whether a page is on the "active pages" or "inactive pages" list. Because this is performed only periodically, it can cause the user OS to have stale page access information. But, all FSVA accesses of file system pages pass through the buffer cache search functions, so we ignore the second case in Linux FSVAs. (Application access to memory-mapped files will cause the user OS's, not the FSVA's, page table entries to be updated.)

Capturing page allocation events in NetBSD is similar to Linux: only one function performs buffer cache allocations. We added a hook to that function. In contrast, NetBSD captures page access frequency differently from Linux. NetBSD lacks a "page accessed" callback in its buffer cache

accesses. As a result, NetBSD solely relies on the periodic scans of page table entries' page-accessed bit. Unlike in Linux, the NetBSD FSVA proxy is thus forced to add a hook to this routine. As described above, relying on the periodic page-access-bit scans leads to the user OS receiving slightly stale page access information. A Linux user OS connected to a NetBSD FSVA may over-aggressively evict buffer cache pages because it lacks up-to-date page-access information.

During startup, OSs allocate bookkeeping structures for every physical memory page. Because the FSVA's memory footprint can grow almost to the size of the initial user OS, we create the FSVA with this maximum memory size. This ensures that the FSVA creates bookkeeping structures for all the pages that it may access. After the OS startup completes, the FSVA proxy returns most of this memory to the VMM.

A subtle unified buffer cache side-effect is that decreasing the number of FSVA free pages affects the dirty page writeback rate. Specifically, the rate of dirty page writeback increases as the amount of free memory decreases. To maintain the same writeback performance, we modified the Linux writeback policy such that when Linux is functioning as an FSVA, the writeback rate depends on the user OS's number of free pages; this value is piggybacked on every request. Because our performance evaluation only uses Linux OSs, we did not make a similar modification to NetBSD.

Although the majority of FSVA memory allocations occur in the buffer cache, FSVA metadata allocations (e.g., for inodes and directory entries) must lead to an increase in the FSVA memory size. Otherwise, memory pressure will cause the FSVA OS to evict buffer cache pages, decreasing performance. Consequently, the FSVA proxy continuously monitors the size of the Linux "slab" — where metadata is allocated — and grows (shrinks) the FSVA as the slab grows (shrinks). The change in slab size is piggybacked on IPC responses, and the user OS changes its size accordingly. OSs handle page-access notification and eviction differently for metadata pages than data pages. As a result, to simplify our implementation, our metadata allocation mechanism lacks eviction backpressure from the user OS. Because our performance evaluation only uses Linux OSs, we did not make a similar

modification to NetBSD.

5.6 Migration

There are two migration facilities in Xen. Ordinary migration consists of saving a VM's memory image in the source host, suspending the VM, transferring the state to the destination host, and resuming that image in the destination host. Live migration [20] minimizes downtime by transferring the majority of a VM's memory image in the background, while the VM executes in the source host. After sufficient memory copying is performed, Xen suspends the VM, copies the remaining pages to the destination host, and resumes the VM.

Migrating a user-FSVA VM pair requires some care, due to the presence of shared memory mappings and in order to preserve live migration's low downtime. A user-FSVA VM pair is migrated like a single VM with three modifications. First, the two VMs' memory images are simultaneously transferred, maintaining the low unavailability of Xen's live migration by synchronizing the background transfer stage. Second, the user-FSVA IPC layer and the shared memory mappings must be reestablished. Third, in-flight requests and responses that were affected by the migration are resent.

We modified Xen's migration facility to atomically transfer two VMs' memory images. To maintain live migration's low downtime, the background transfer of the two memory images and the suspend/resume events are synchronized. For example, if the FSVA's memory image is larger than the user VM, we delay the suspension of the user VM until the FSVA is ready to be suspended. This preserves the minimal suspend time for the user VM. Because the user VM depends on the FSVA, the user VM is suspended first and restored second.

When a VM is resumed, its IPC layers to other VMs are broken. Thus, the user OS and FSVA proxies must reestablish their IPC layer. This involves recreating the shared-memory ring and communication channels. In addition, shared memory mappings must also be reestablished. A shared memory mapping in Xen depends on the two VMs' numerical IDs and the physical

page addresses. After migration completes, these values will likely be different. Consequently, all shared memory mappings between the two VMs must be reestablished. To support this operation, the FSVA proxy maintains a list of all shared memory pages. After the user VM is resumed, the user OS proxy performs a `restore_grants` IPC to retrieve this list from the FSVA. The user OS proxy leverages Xen's batched hypercall support to speed up the reestablishment of the shared memory pages.

After a user VM is resumed, applications may attempt to access a memory-mapped page whose mapping has not yet been restored. This would cause a segmentation fault. To avoid this race, we made a single-line change to the Xen migration code to zero-out user VM page table entries that point to another VM. So, application attempts to access the page will cause an ordinary page fault into user OS, and the user OS proxy will block the application until the page's mapping is reestablished.

Because the user-FSVA IPC layer is broken during migration, in-flight requests and responses must be resent. To enable retransmission, the user OS retains a copy of each request until it receives a response. To ensure exactly-once IPC semantics, unique request IDs are used and the FSVA proxy maintains a response cache. The ID is a function of the request's position in the IPC ring. Data operations are assumed to be idempotent and hence the response cache does not contain data blocks. The FSVA proxy garbage collects a response upon receiving a new request with an ID matching the cached response's original request's position on the IPC ring.

6 Evaluation

This section evaluates our FSVA prototype. First, it describes examples of using FSVAs to address file system portability. Second, it quantifies the performance and memory overheads of our FSVA prototype. Third, it illustrates the efficacy of the inter-VM unified buffer cache and migration support.

6.1 Experimental setup

Experiments are performed on a dual quad-core 1.86 GHz Xeon E5320 machine with 8 GB of memory, a 10K rpm 146 GB Seagate Cheetah ST3146755SS disk connected to a Fusion MPT SAS adaptor, and a 1 Gb/s Broadcom NetXtreme II BCM5708 Ethernet NIC. The NFS server is a single quad-core 1.86 GHz Xeon E5320 machine with 4 GB of memory, a 10K rpm 73 GB Seagate Cheetah ST373455SS disk using the same Fusion SAS adaptor and Broadcom NIC, running the Linux in-kernel NFSv3 server implementation.

Xen 3.4-unstable was used. Linux VMs run 64-bit the Debian "testing" distribution, with either our modified 2.6.18 kernel (based on the Xen-maintained Linux kernel tree) or our modified 2.6.28 kernel (based on the vanilla Linux repository). We compiled the Linux kernels with gcc 4.3.3, without debugging symbols or checks. NetBSD VMs run 64-bit NetBSD 5.99.5, compiled with gcc 4.1.3 with debugging symbols enabled.

By default, the ext2 and ext3 file systems randomly allocate block groups for top-level directories. This caused significant variance in our results — as much as 15% across runs. To avoid this variance, we used the ext2 and ext3

`oldalloc` mount option; this forces a deterministic, but slower, block group allocation algorithm.

When running benchmarks on a local file system, the file system used a 108 GB raw disk partition. The NFS server exported an 18 GB ext2 partition (mounted with the `oldalloc` option).

Unless otherwise noted, each VM was given 2 GB of memory. For FSVA experiments, the inter-VM unified buffer cache allowed us to specify a total of 2 GB for both the user VM and the FSVA; the user-FSVA VM pair do not benefit from extra caching.

6.2 Case studies: portable file system implementations

The efficacy of the FSVA architecture in addressing the file system portability problem is demonstrated with two case studies: one inter-OS and one intra-OS.

Linux user using NetBSD LFS. Linux does not include a log-structured file system [82] (LFS) implementation. There are stale third-party in-kernel and user-level implementations, but they are either unstable, not full-featured, or have not been ported to modern Linux kernel versions.

NetBSD includes an in-kernel LFS implementation, derived from Seltzer's FreeBSD LFS re-implementation [89]. Using an FSVA, a Linux 2.6.28 user OS can use the unmodified NetBSD LFS implementation (see Figure 6.1). The user-level LFS cleaner also does not require modifications; it runs in the NetBSD FSVA.

We ran a random I/O benchmark in the Linux user OS, using a 200 MB test file and a 512 byte write unit size. When running over the NetBSD LFS FSVA, the benchmark achieved 19.4 MB/s. In contrast, when running over a Linux ext3 FSVA, the benchmark only achieved 0.44 MB/s. Such improved random write performance is the hallmark of the LFS approach. FSVAs enable Linux users with such workloads to benefit from LFS's efficiency without forcing them to switch to NetBSD or developers to port LFS to Linux.

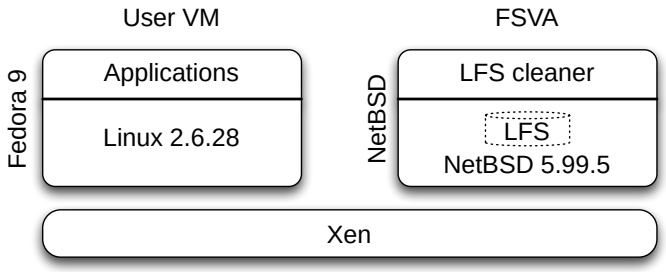


Figure 6.1. NetBSD LFS case study.

Linux 2.6.18 user using Linux 2.6.28 ext4fs. The Linux 2.6.28 kernel (released in December 2008) includes a new local file system: ext4. In contrast to its widely-used ext3 predecessor, ext4 adds extents, delayed allocation, and journal checksumming.

Using FSVAs, a user OS running a Linux 2.6.18 kernel (released in September 2006) can use a Linux 2.6.28 ext4 FSVA. Compared to ext3, the ext4 FSVA provided over a 4 X improvement in Postmark performance; the Postmark benchmark is described in the next subsection. Thus, FSVAs enable a user with a two year-old Linux kernel to gain the benefits of ext4 immediately, without being forced to upgrade.

6.3 Macrobenchmarks

To quantify FSVA overheads, we use three file system-intensive macrobenchmarks: Postmark, IOzone, and a Linux kernel compilation. To focus on FSVA overheads, both the user OS and the FSVA used an identical OS: Linux with a 2.6.28 kernel. Otherwise, differences in internal OS policies add variables to the comparisons. For example, eviction and writeback policies are different in the 2.6.18 and 2.6.28 kernels, and NetBSD performs fewer permission VFS calls than Linux due to its whole-pathname name cache, in contrast to Linux's per-pathname-component name cache. Using the same OS version in the user OS and FSVA avoids having these policy and implementation differences muddle our analysis.

The macrobenchmarks were executed over four file systems: ext2, ext3, NFS, and ReiserFS. Six system configurations were used. "baremetal" denotes the OS running directly on the hardware without a VMM. "domU" denotes the OS running as a paravirtualized Xen guest. In both cases, the file system executes "natively" in the OS kernel. "FSVA-same-core" denotes an FSVA sharing the same CPU core as the user VM and using Xen event channels for signaling (§5.3.2); each IPC involves two VM context switches. In the next three configurations, the user OS and FSVA are executed on separate cores, and we explore the performance of our three different signaling mechanisms. "FSVA-diff-core-events" uses Xen event channels, "FSVA-diff-core-polling" uses polling, and "FSVA-diff-core-mwait" uses our new `mwait`-based inter-VM signaling hypercall (§5.3.2).

Of the six system configurations, we are most interested in the performance difference between "domU" and "FSVA-diff-core-mwait". This difference is the FSVA architecture's overhead when both VMs concurrently execute on different cores. We envision a VMM scheduler that gang-schedules both VMs during file system-intensive periods. We are also interested in the performance difference between "baremetal" and "domU"; this is the performance overhead of virtualization. We expect processor, VMM, and OS improvements to decrease this overhead over time, as virtualization continues its increasing adoption.

Each experiment was run three times; means and standard deviations are shown. Before each experiment, the file system partition was reformatted and caches were flushed.

Postmark. The Postmark benchmark measures performance for small file workloads akin to e-mail and newsgroup servers [51]. It measures the number of transactions per second, where a transaction is either a file create or delete, paired with either a file read or append. Files are created with randomly-varying sizes varying from 500 bytes to 9.77 KB. Appends use access sizes that randomly vary from 1 byte to the file size. Reads access the entire file. Default parameters were used, except for benchmark sizing: we increased the benchmark size to 50,000 files, 50,000 transactions, and 224 subdirectories.

Figure 6.2 shows that the use of virtualization and polling- or `mwait`-based IPCs results in less than a 10% performance reduction for FSVAs, for all tested file systems except ext2. Virtualization introduces a 10% overhead in ext2. FSVAs, using polling- or `mwait`-based IPC, introduce an additional 7% overhead when compared to a native in-kernel ext2 system. The lack of journaling in ext2 gives it the fastest absolute performance and thus Xen and FSVA overheads are more prominent.

For NFS, all configurations exhibit less than 4% reduction. For the local file systems, though, the non-polling/`mwait` FSVAs suffer more overhead. For ReiserFS, there is a 6.7% slowdown when going from baremetal to virtualization, and an additional 6.9% overhead for the single-core and events-based IPC FSVA configurations. For ext2 and ext3, the single-core and events-based IPC configurations introduce 21–24% reductions in throughput compared to the domU system. This high overhead is due to the high frequency and cost of FSVA IPCs when polling/`mwait` are not used. Thus, separate cores and polling/`mwait` is essential for achieving low overheads during in-cache file system-intensive operation.

IOzone. The IOzone benchmark supports a wide range of sequential/random workloads [27]. We used IOzone to measure sequential I/O performance. A 10 GB file was sequentially written and read, using 64 KB record sizes. Because the file was much larger than the VM memory size, the numbers reflect out-of-cache performance.

Figure 6.3 shows that for all file systems, there was less than 2.5% difference among the various configurations. These results indicate that virtualization and the use of FSVAs do not impact streaming I/O throughput, even when the user VM and FSVA share a single CPU core. This supports our hypothesis that the need for multiple cores can be avoided for out-of-cache operation, in which the FSVA overhead is hidden by device access time (§8.1).

Linux kernel build. This benchmark consists of building the Linux 2.6.28 kernel. The kernel archive was copied to the file system, unarchived, and compiled. Approximately 1000 source files were compiled for our specific kernel configuration.

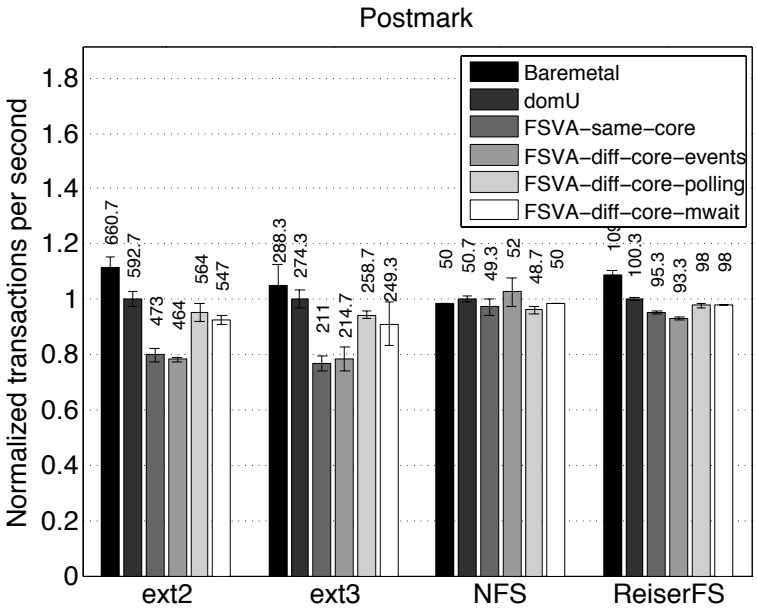


Figure 6.2. Postmark results, normalized to domU.

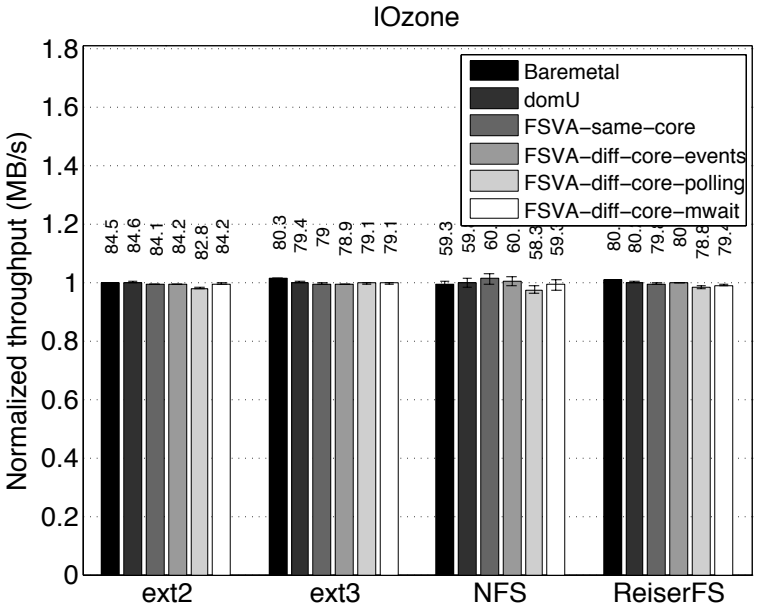


Figure 6.3. IOzone results, normalized to domU.

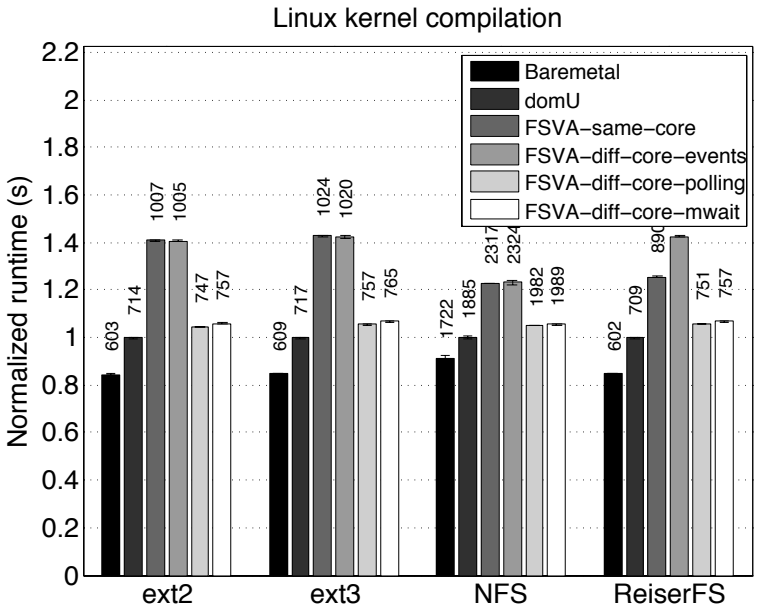


Figure 6.4. Linux kernel compilation runtime, normalized to domU.

Figure 6.4 shows the results. Virtualization adds substantial overhead (6–18%) to the Linux kernel compilation. We suspect that frequent program execution (for the custom Linux build scripts and compiler invocations) leads to many memory-management hypercalls that cause this overhead.. When using FSVAs, the overhead varies significantly based on the configuration. With a separate CPU core and polling or `mwait`, the overhead is $\leq$7%. For the single CPU core and the no-polling configurations, 20%–40% slowdowns occur. The culprit for these slowdowns is frequent `permission` IPCs. For example, for the ext3 case, `permission` IPCs account for 60% of the 9,177,610 IPCs. For all file systems tested, the `permission` VFS handler is very simple: it calls the generic OS access control handler that compares the Unix user ID with the file owner ID and mode permission. Our design decision to pass all VFS calls (§4.2.1) highlights the IPC overhead for such simple IPCs. Executing the permission checks in the user OS would significantly reduce this overhead, as discussed in §6.5.

6.4 Microbenchmarks

To understand the causes of the FSVA overhead, we used high-precision processor cycle counters to measure two sets of operations: IPC layer costs and VFS microbenchmarks.

Table 6.1 lists the IPC layer costs, showing the medians of ten runs. Before explaining the performance of the different IPC types, it is necessary to first describe the building block operations. A "null hypercall" measures the round-trip cost of a VM-VMM call: it has similar cost to an OS system call. The "send event" operation refers to sending a notification to another VM using Xen's event channels.

There are four costs to sharing a memory page between VMs. First, the source VM creates a grant that covers the page. Second, the destination VM makes a "map grant" hypercall. Third, to remove the shared memory mapping, the destination VM makes an "unmap grant" call. Fourth, the source VM destroys the grant. Note that it is more efficient to "share" up to five 4 K pages through memory copies over a dedicated staging area than through using shared memory (4 us versus 4.75 us). However, for larger transfer sizes, the grant mechanism is faster due to amortization of the hypercall cost.

A traditional Xen IPC requires two event notifications, each consisting of an inter-processor interrupt (IPI) and two thread switches (§5.3.2). Those four operations correspond to 18 μs of the 21.21 μs null IPC latency we observed. The remainder of the IPC latency goes towards locking the shared IPC ring, copying the request and response data structures onto the ring, and other miscellaneous operations.

When inter-VM signaling is performed using polling or our new `mwait` hypercall, the null IPC latency drops to 4.04 μs and 4.34 μs, respectively. The extra latency for the `mwait`-based IPC is due to the cost of the hypercall that wraps the privileged `mwait` instruction. Note that, when the two VMs are pinned to the same core, Xen avoids sending an IPI and only performs a VM context switch, leading to a slightly faster IPC (16.70 μs versus 21.21 μs).

(The OS thread switches still occur because the OS still executes its interrupt-handling routine once the VM is scheduled.)

Polling and `mwait` achieve a 4 × reduction in IPC latency compared to the traditional Xen IPC. This improvement is significant for fast and frequent VFS operations such as in-cache metadata lookups.

Table 6.2 shows VFS operation latencies, run over ext2 for the system configurations described in §6.3. The results are the mean of 100 consecutive runs over an in-cache workload. Most VFS operations suffer signficant slowdowns when running over the FSVA due to the IPC layer costs. For example, a `getattr` has a latency of 0.09us in a native file system, 26.25 us when the user OS and FSVA share a core, and 5.65 us when dedicated cores and `mwait` are used. Thus, even with dedicated cores, there is still a factor of 50 X latency increase. This is unavoidable: the ext2 `getattr` handler simply copies a file's attributes (that are already in memory), and so the operation is very fast. The IPC layer cannot approach this level of performance.

Fortunately, in many file system workloads the disk is accessed, either for logging or because the data does not fit in memory. Disk access latencies dwarf the IPC layer overhead. Thus, the macrobenchmarks in §6.3 suffer an overhead of 0%–40%, not 5000%.

6.5 Relaxing the "pass all VFS calls" principle

Our design goal of maintaining file system semantics for unmodified file system implementations forces all VFS calls to be sent to the FSVA (§4.2.1). This leads to a high frequency of IPCs in which the IPC latency dominates the VFS operation's cost, significantly affecting application performance. The macrobenchmark results in §6.3 demonstrate that reasonable performance can be achieved if a processor core is dedicated to the FSVA, enabling low inter-VM IPC latency (§5.3.2). But, dedicating a core to the FSVA may not always be feasible or desirable. This section explores the effect of relaxing the "pass all VFS calls" requirement.

When the user VM and FSVA execute on the same core, the highest overhead in the macrobenchmarks results occurred for the Linux kernel

Operation	**Latency (μs)**
Null hypercall	0.24
Send event (hypercall+IPI)	2.09
Create grant	0.21
Destroy grant	0.36
Map grant	1.99
Unmap grant	2.19
4 KB memcpy	0.80
Thread switch	3.52
Null IPC (same-core, event)	16.70
Null IPC (diff-core, event)	21.21
Null IPC (diff-core, polling)	4.04
Null IPC (diff-core, mwait)	4.34

Table 6.1. IPC layer latencies.

Operation	**domU**	**FSVA same-core**	**FSVA diff-core events**	**FSVA diff-core polling**	**FSVA diff-core mwait**
create	2.20	22.24	31.24	6.95	7.89
getattr	0.09	26.25	28.50	4.61	5.65
lookup	19.70	40.45	53.45	19.89	25.68
mkdir	742.60	783.0	730.8	777.0	746.0
open	0.08	18.05	27.20	4.90	5.61
permission	0.10	20.40	29.82	6.22	6.97
read (4 KB)	2.24	31.59	36.57	9.35	10.32
release	0.17	19.36	29.72	5.10	5.91
rename	1.50	21.03	32.09	6.95	7.69
rmdir	1.14	23.32	28.13	5.66	6.49
setattr	0.22	22.62	28.25	4.84	5.46
unlink	0.88	20.56	29.67	6.07	6.99
write (4 KB)	4.30	31.21	39.04	11.37	12.24

Table 6.2. VFS operation latencies in (μs).

compilation over ext2 and ext3. Compared to the native file system ("domU"), the FSVA overhead was 40%. To understand the source of this overhead, Table 6.3 lists the per-VFS operation IPC counters and timers for the ext2 run. Of the 9,177,610 IPCs, 60% are `permission` IPCs, and 53% of the user OS proxy time is accounted for by `permission` IPCs.

The ext2 `permission` handler calls the generic OS access control function, which compares the Unix user ID with the file owner ID and mode permission. Given the simplicity of this check, the IPC overhead dwarfs the VFS operation execution time. Fortunately, the Unix access control check is common to all Unix OSs. Thus, for ext2, we can eliminate the `permission` IPCs because the user OS `permission` handler is sufficient.

With the `permission` IPC eliminated — the user OS proxy calls the user OS generic `permission` handler — the Linux kernel compilation benchmark overhead over ext2 is reduced from 40% to 11% when the user VM and FSVA share a processor core.[1] This performance improvement demonstrates that relaxing our design principle of passing all VFS calls can alleviate the need to dedicate a processor core to the FSVA. Although this requires file system modifications (e.g., to indicate that a user OS VFS handler is sufficient or to add cache invalidation handlers if user OS caching is permitted), the performance improvements make this worthwhile.

In addition to avoiding the `permission` IPC, Table 6.3 shows that significant, though lesser, performance improvements can also be gained by caching data (`read`), metadata (`getattr`), and delaying or batching user OS calls (`release` – the *close file* VFS handler). However, in contrast to the `permission` IPC, this would require more extensive file system changes and will not have as significant of an impact as avoiding the `permission` IPC.

[1] This overhead is lower than the overhead calculated by subtracting the `permission` timer in the user OS proxy (185s; see Table 6.3) from the original time (1007s; see Figure 6.4). We suspect that this is due to a reduction in TLB and other processor cache misses resulting from a decrease in the number of IPCs and, thus, the number of VM context switches.

Operation	Total time (s)	Count
permission	185.42	5663803
read	30.99	557061
release	28.38	776795
getattr	24.90	784716
open	23.62	776795
write	19.82	158261
mkdir	13.44	1892
llseek	7.43	161621
readlink	4.39	130155
lookup	1.40	37455
create	1.35	30848
readdir	1.07	8293
unlink	0.95	30846
setattr	0.89	27436
ubc_flush	0.76	12100
map	0.62	15835
rmdir	0.09	1892
rename	0.06	1773
write_inode	1.1×10^{-3}	27
dentry_release	6.2×10^{-4}	4
symlink	9.4×10^{-5}	2

Table 6.3. Per-VFS operation elapsed time and frequency counters for the Linux kernel compilation benchmark over ext2.

6.6 Memory overhead

FSVA memory overhead has two components: memory for the FSVA OS image and memory for duplicated metadata. The particular values for this memory overhead vary depending on the particular OS image and the amount of metadata in use. As concrete examples, we report the memory overhead when running the macrobenchmarks reported in §6.3.

The Linux 2.6.28 FSVA uses 72 MB of memory for the OS image. Our FSVA proxy sets aside 64 MB of memory for an initial extra reservation. Then, during benchmark execution, we observed 112–136 MB of additional memory allocated for metadata. Thus, the total memory overhead was 248–272 MB.

FSVA memory overhead can be reduced in two ways. First, the FSVA Linux kernel configuration can be fine-tuned to remove functionality that is unnecessary for an FSVA. For benchmarking purposes, we used the same Linux 2.6.28 kernel for all experiments. But, when running as a Xen paravirtualized VM and only supporting a limited application (the file system), the FSVA kernel can be trimmed down. Second, as described in the §5.5, we currently do not put pressure on the size of the metadata allocated in the FSVA. Improvements to our unified buffer cache implementation could trigger metadata eviction in the FSVA.

6.7 Unified buffer cache

To demonstrate the efficacy of our inter-VM unified buffer cache, we ran an experiment with an application alternating between file system and virtual memory activity. The total memory for the user VM and FSVA is 1 GB. Both VMs are created with 1 GB of memory. After the user and FSVA kernel modules are loaded, the FSVA returns most of its memory to Xen, thereby limiting the overall memory usage to slightly over 1 GB.

Figure 6.5 shows the amount of memory each VM consumes. Starting with a cold cache, the application reads a 900 MB file through memory-mapped I/O. This causes the FSVA's memory size to grow to 900 MB, plus its overhead. The application then allocates 800 MB of memory and touches

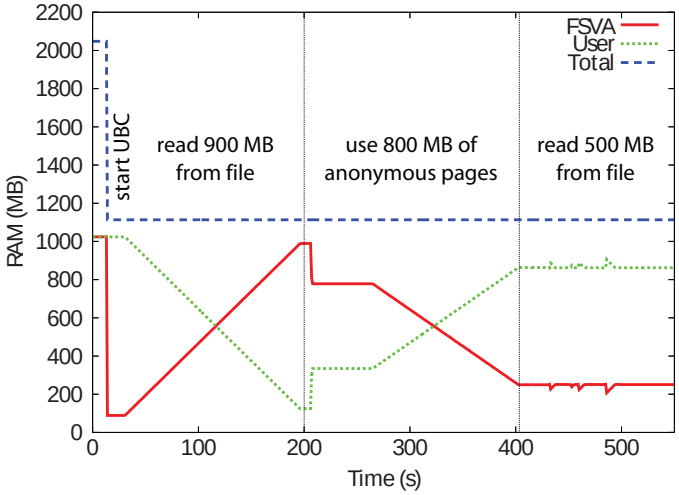


Figure 6.5. **Unified buffer cache demonstration.** This figures shows the amount of memory consumed by the user and FSVA VMs. As applications shift their memory access pattern between file system and virtual memory usage, the unified buffer cache dynamically allocates memory among the two VMs while maintaining a constant total memory allocation.

these pages, triggering Linux's lazy memory allocation. As the allocation proceeds, the user VM evicts the clean file system pages to make room for the virtual memory pressure. These evictions trigger UBC IPCs to the FSVA that return memory to the user VM. Initially, Linux batches the evictions, causing the non-linear drop in the beginning of the second phase.

In the third phase, the application performs a 500 MB ordinary read from a file. This requires file system pages to stage the data being read. Because the application still contains an 800 MB allocation, and swapping is turned off for this experiment, the virtual memory pages cannot be evicted. The result is that only the remaining memory (just over 200 MB) can be used to stage reads; the unified buffer cache constrains the FSVA to this size. Page eviction batching is responsible for the dips in the figure.

This experiment demonstrates the effectiveness of our unified buffer cache implementation in two ways. First, a single eviction policy (in the user VM) based on virtual memory and file system cache usage is preserved. Second, the combined memory usage of the two VMs stays constant.

6.8 Migration

To evaluate the FSVA's effect on unavailability during live migration, we wrote a simple benchmark that continuously reads a memory-mapped file. Every microsecond, the benchmark reads one byte and sends a UDP packet containing that byte to another machine. The second machine logs the packet arrival times, providing an external observation point for measuring the slowdown introduced by migrating the user-FSVA VM pair.

To establish baseline live migration performance, we ran our benchmark against the root NFS filesystem of a single VM with 512 MB of memory. During live migration, the unavailability period was 0.29 s. We then repeated this test against the same file system exported from an FSVA to a user VM. The two VMs' memory allocation was set to 512 MB plus the overhead of the FSVA's operating system, which was approximately 92 MB. Unavailability increased to 0.51 s. This increase is caused by the extra OS pages that must be copied during the suspend phase and the overhead of restoring our IPC layer and shared memory pages. We believe this overhead is relatively independent of the overall memory size. But, we were unable to run migration experiments with larger memory footprints due to limitations in the size of Xen's preallocated shadow page tables that are used during migration.

7 Experiences

This chapter describes our experiences in implementing the FSVA prototype. It describes the changes to the FSVA interface that occurred during FSVA ports and discusses our expectations for future ports. It also describes implementation pitfalls for others interested in running a file system in its own VM.

7.1 Porting experience and expectation for future ports

For the FSVA approach to succeed, the FSVA interface must be stable. Earlier, we argued that a minimal VFS-like FSVA interface can remain stable while supporting different Unix OSs, because inter-OS differences tend to occur in internal OS implementation (e.g., memory management) (§3.3.1). This section describes the effects of porting the FSVA prototype to different OSs and VMMs on the FSVA interface, and our expectation for future ports. In particular, the salient question is whether the FSVA interface remains stable across ports to new OSs or new OS versions. If not, FSVAs merely shift the original changing-interfaces problem, and OS vendors will not likely adopt the user OS and/or the FSVA proxies.

The initial FSVA prototype was implemented using the Linux 2.6.18 kernel. The FSVA proxy was then ported to Linux 2.6.25, followed by a port of the user OS proxy to Linux 2.6.25. Subsequently, both proxies tracked the Linux 2.6.26, 2.6.27, and 2.6.28 kernel releases. In addition, Karan Sanghi implemented[1] the FSVA proxy for NetBSD 5.99.5 [85]. Together, the Linux

[1]Porting the proxies across different OS versions requires relatively minor changes. In contrast, porting the proxies to a different OS requires a complete rewrite due to the

and NetBSD ports provide insight into the viability of a stable FSVA interface across intra-OS and inter-OS ports, respectively.

Porting the Linux FSVA proxy from the 2.6.18 kernel to the 2.6.25 led to a significant change in how files are identified. In our initial implementation, we used inode numbers to identify files. On response to a `lookup` operation, the FSVA proxy would return the file's inode number. The user OS proxy would pass the inode number to the FSVA proxy in future requests, and the FSVA proxy would retrieve the corresponding file by calling the generic VFS inode-number-to-inode mapping function. But, this function was removed after the Linux 2.6.18 kernel release. This forced us to change how files are identified: we switched to using pointers to FSVA in-memory data structures (§5.2). The resulting FSVA interface made fewer requirements from the FSVA OS: instead of requiring the FSVA OS to provide an inode-number-to-inode mapping function, we relied on using data structure addresses. Subsequent ports to the Linux 2.6.26–2.6.28 kernels as well as to NetBSD demonstrated the genericness of our new file identification mechanism. Thus, although the port from Linux 2.6.18 to 2.6.25 led to an FSVA interface change, this was a "one-off" interface generalization to fix a naive initial design assumption. Intuitively, given that the current file identification only relies on data structure memory addresses, the new mechanism will work for any OS.

Porting the Linux user OS proxy from Linux 2.6.18 to 2.6.25–2.6.28 required no FSVA interface changes. Although changes occurred in internal Linux interfaces – for example, the syntax of internal caches and VFS handlers that support memory-mapping changed – we were able to address these changes in the the user OS proxy without affecting the FSVA interface. This validates our intra-OS FSVA interface stability argument (§3.3.1).

Implementing the FSVA proxy for NetBSD led to two changes to the FSVA interface. First, the initial FSVA prototype assumed that the user OS and FSVA share the same size for the directory entries returned by the `readdir` system call. However, the POSIX standard specifies that the data structure size is OS-specific [30]. Consequently, we revised the FSVA interface such

substantial differences among internal OS interfaces. We stress this difference in porting effort by referring to an inter-OS FSVA port as a new implementation.

that the user OS passes its directory entry size in the `readdir` operation [85]. Second, because NetBSD delegates directory entry caching to the file system, in contrast to Linux which performs directory entry caching in the generic VFS layer, the FSVA `lookup` interface changed. Specifically, in our initial FSVA interface, a negative `lookup` result, indicating a file's nonexistence, returned a valid directory entry pointer. This reflected the semantics of the Linux generic `lookup` behavior. In contrast, the generic NetBSD `lookup` function returns a null pointer. Consequently, for VFS operations that may refer to the result of a previous negative `lookup` operation, such as `create`, NetBSD requires the filename to be provided, in contrast to Linux in which a pointer to the negative directory entry is passed. The FSVA interface was modified to pass a directory entry pointer if one was returned by a previous `lookup`; otherwise, the filename was passed.

Although the NetBSD port required two FSVA interface changes, we again argue that these changes reflect "one-off" initial Linux-specific assumptions. In particular, almost any non-Linux port would have exposed the directory entry size issue. Additionally, the differences in referring to negative directory entries have long been known [50]: a careful study of earlier attempts at a universal VFS interface would have exposed the issue. Thus, we argue that the FSVA interface will converge to a stable universal interface after a small number of OS ports expose initial OS-specific assumptions.

To understand the effect of future OS ports on the FSVA interface, we studied the Solaris VFS interface [60]. Path lookup in Solaris is very similar to NetBSD — the file system is responsible for managing the directory entry cache and negative directory entries. Given that the NetBSD port required significant FSVA interface changes to handle pathname lookup, we expect that a Solaris port would not affect FSVA file identification. Furthermore, although Solaris has several VFS calls that implement reader/write inode locking, we believe that a Solaris user OS proxy could implement these operations locally without involving the FSVA. This would leave the FSVA interface unaffected. Note that our single user OS per FSVA design ensures that no other OS would be accessing these inodes. In terms of data structures, Linux, NetBSD, and Solaris all share very similar inode data structures. Although the Solaris

directory entry size is different from Linux, our NetBSD-port fix for passing the user OS directory entry size in the `readdir` operation handles Solaris. Overall, we believe that our NetBSD port sufficiently generalized our FSVA interface such that a Solaris port would require no changes to the FSVA interface.

Our FSVA design eschewed a single FSVA interface for both Unix and Windows OSs. We argued that separate FSVA interfaces for Unix and Windows would encourage OS vendor adoption by simplifying the user OS and FSVA proxies (§4.2.3). Nevertheless, we now describe our expectations if a single FSVA interface was used for both Unix and Windows.

Several papers describe porting Unix research file systems to Windows. For example, Coda was ported to Windows 95/98/NT [11], Arla (an AFS clone) was ported to Windows NT [3], and the Secure File System was ported to Windows NT [79]. These ports demonstrate that a Unix file system can retain its semantics for Windows clients. This is not surprising because Unix has a smaller file system interface than Windows (e.g., Windows allows access control lists and byte-range locking). Consequently, we expect that a Windows user OS proxy implementation is unlikely to require changes to the FSVA interface.

In contrast, allowing Unix user OSs to access Windows FSVA is likely to require FSVA interface changes. The FSVA interface must support Windows' rich access control lists, fine-grained locking, directory notifications, and different naming schemes. Unix applications could access these features through FSVA-specific `ioctl`s. Furthermore, Unix FSVA proxies would need to implement these now-standard FSVA features to support Windows users, complicating Unix FSVA proxies.

In addition to maintaining a stable interface across OS ports, interface stability across different VMM ports is also important. We briefly describe our experience in porting the Linux user OS and FSVA proxies from Xen to VMware Workstation; full details are in Saurabh Shah's Masters report [90]. Although differences exist among Xen's and VMware's bootstrapping protocols (e.g., VM discovery) and interfaces (e.g., shared memory hypercalls), these differences are easily localized in the user OS and FSVA proxies. As a

result, the FSVA interface was not changed during the VMware port.

7.2 Lessons for others running a file system in its own VM

Several other research projects have explored running a file system in its own VM, for a variety of reasons such as extensibility [76], performance [58, 112], and security [59, 107]. Regardless of the motivation and specific design choices, there are common pitfalls when running a file system in its own VM. These issues caught us by surprise. As warning to others, we describe three issues: memory-mapped I/O, unified buffer cache, and performance comparisons.

To support memory-mapped I/O, Unix OSs have a VFS handler that read a file system page into memory. In the Linux user OS proxy, this VFS handler maps the relevant FSVA page into the user OS's address space using VMM shared-memory hypercalls (§5.3.1). Application-level memory map system calls do not typically involve the file system; once the file system reads a page into the OS address space, the OS is responsible for setting user-level page table pointers to refer to the relevant page.

Xen requires a special hypercall to allow user-level page table pointers to refer to another VM's page. Consequently, our prototype required several changes to the Linux virtual memory subsystem to allow user-level memory-mapped I/O to point to an FSVA page. We added hooks to the existing OS "set page table pointer" routines that enabled the user OS proxy to make the special Xen hypercall. These changes were responsible for the majority of the Linux OS changes (Table 5.1). Furthermore, almost every port to a new Linux kernel release broke this functionality due to changes in the virtual memory subsystem. By far, fixing this functionality was the most challenging aspect of porting the user OS proxy to a new Linux kernel version. On the one hand, we were pleased that the fixes never affected the FSVA interface. On the other hand, we usually spent several days debugging obscure page table pointer errors. Note that this issue may be Xen- or paravirtualization-specific.

Memory is a scarce resource. To make the best use of memory, a unified buffer cache avoids data duplication and optimally divides memory between the file system buffer cache and the virtual memory page cache [37, 91]. But,

when a file system runs in its own VM, the existing unified buffer cache functionality is broken (§4.4.3). None of the previous systems addressed this problem, leading to increased memory usage. Furthermore, although VMM-based page deduplication [103] avoids data duplication, it does not enable a single eviction policy[2].

This dissertation describes an inter-VM unified buffer cache implementation that requires few OS changes (Table 5.1). Our design is not FSVA-specific; other file system VMs can use our approach. Furthermore, although our design dedicates an FSVA for each user OS (§4.2.2), our unified buffer cache design will mostly work even in a shared FSVA: each user OS can maintain its own eviction policy. In a shared file system VM design, it will be necessary to recognize which user OS is responsible for page additions and accesses in the file system VM. We expect that tagging directory entries with the corresponding user OS identifier should be sufficient.

It is important to accurately measure the performance overhead of file system VMs: high overhead can affect a system's viability. Unfortunately, our initial FSVA performance measurements were flawed. We compared FSVA performance with native file systems as soon as we observed a correct macrobenchmark execution — as judged by the benchmark running to completion and/or generating the correct output. But, this approach fails to take into account the performance overhead of VFS operations that were mistakenly *not* sent to the FSVA. For example, our initial user OS proxy neglected to send the `permission` VFS call, used for access control, to the FSVA. NFS often performs network operations in response to a `permission` check. Consequently, this led to FSVA performance incorrectly exceeding native file system performance.

To catch overlooked VFS IPCs, we added counters to all VFS operations. This allowed us to compare, say, how many times the NFS `permission` VFS handler executed when running a benchmark over an NFS FSVA and over a native NFS. Yet, FSVA NFS performance still unexpectedly exceeded native NFS performance. The culprit was our neglect to pass certain OS flags

[2]When the user OS evicts a page, the VMM breaks the deduplication, leaving the file system VM with a copy of a page that should be evicted from memory.

for the `open` and `create` operations. When these flags were set, the NFS handlers perform multiple network operations during a single VFS operation. Thus, counting VFS operations was insufficient. We switched to performing network-level protocol traces for NFS and counting block-level reads/writes for local file systems.

8 Conclusion

FSVAs offer a solution to the file system portability problem, leveraging the virtualization and multicore processor technology trends. A file system is developed for one OS and bundled with it in a preloaded VM. Users run their preferred OS and use the FSVA like any other file system. The file system is isolated from both kernel- and user-space differences in user OSs, because it interacts with just the single FSVA OS version. In contrast to previous approaches, FSVAs accommodate existing file system implementations and provide more robust isolation from the user OS.

We presented an FSVA design with minimal performance overheads and no visible semantic changes for the user. File system semantics are maintained without file system modifications, thus supporting legacy file systems implementations. OS and virtualization features, such as a unified buffer cache and VM migration, are maintained, thus encouraging user adoption.

An FSVA prototype demonstrates efficient performance using multicore processors. Case studies and other experiments demonstrate that the FSVA approach works for a range of unmodified file system implementations across distinct OSs. Few changes are made to the OS and VMM, thus encouraging vendor adoption.

For the FSVA approach to be viable, the user OS and FSVA proxies must be maintained by OS vendors. Our design encourages vendor adoption through design principles that simplify the proxies' implementation and ensure a stable FSVA interface.

8.1 Future work

Our design goal of supporting unmodified file systems comes with a cost: it prevents caching in the user OS. This leads to higher performance overhead. Although multicore processors mitigate this overhead, this increases resource utilization. An auto-negotiation scheme can simultaneously accommodate unmodified legacy file system implementations and provide more efficient operation for FSVA-optimized file systems by reducing the number of IPCs. For example, adding an optional cache invalidation protocol to the FSVA interface may alleviate the need for multicore processors even during in-cache file system-intensive operation. Similarly, file systems could indicate that a user OS VFS handler can directly handle an operation (see §6.5).

Adaptive gang-scheduling can maintain efficient performance, without requiring constant gang-scheduling of the user OS and FSVA. A VMM- or proxy-based scheduler can detect in-cache file system-intensive operation and gang-schedule a user OS and its FSVA(s). During light file system usage or out-of-cache file system operation (in which the IPC latency is dwarfed by device access time), the user OS and FSVA can be scheduled on the same processor core [33, 72].

Although inter-VM control transfer and notification is the FSVA performance bottleneck, there is a significant cost to frequent shared memory hypercalls. Given the trust model between the two OSs, a shared address space is appropriate and avoids the overheads of shared memory hypercalls and the resulting TLB flushes. Fido [14] implements a single address space among an arbitrary number of VMs; we can borrow its techniques.

A stable FSVA interface is crucial for encouraging OS vendor adoption. Our port to different Linux kernel versions and NetBSD required some changes to the FSVA interface. But, we believe that the FSVA interface is converging to a stable interface. Porting the proxies to other Unix OSs, such as AIX or Solaris, will demonstrate FSVA interface stability. Similarly, a Windows port will shed light on whether a universal FSVA interface is viable or if an FSVA interface for each major OS is more appropriate.

FSVAs execute file system implementations separately from the rest of the

user OS. This is in the spirit of the Mach multi-server vision [95]. But, FSVAs exploit virtualization to reduce the significant upfront effort traditionally imposed by microkernels. This separation does not have to be confined to just file systems. For example, networking stacks and device drivers would benefit from this separation, delivering the promises of microkernels: greater implementation flexibility and more robust OSs [76].

To encourage the separation of OS components into VMs, two software and hardware changes are required. First, VMMs should treat a set of VMs as a single logical unit. This provides efficiency (e.g., a single address-space, appropriate for this trust model, can eliminate data copying and/or shared memory hypercalls) and maintains virtualization features (e.g., preserving migration support without OSs' involvement). Second, processors and VMMs should provide low-latency inter-VM communication primitives (e.g., §5.3.2). This avoids efficiency being an obstacle to fine-grained OS-component separation.

9 Glossary

Term	Definition
FS	File system
FSVA	File system virtual appliance
IPI	Inter-processor interrupt
Hypercall	A synchronous software traps from a VM to the VMM
Hypervisor	See *VMM*
SLOC	Source lines of code
UBC	Unified buffer cache
VFS	Virtual File System
VM	Virtual machine
VMM	Virtual machine monitor

Table 9.1. Terminology

Bibliography

[1] Abd-El-Malek, M., Wachs, M., Cipar, J., Ganger, G. R., Gibson, G. A., and Reiter, M. K. 2008. File system virtual appliances: Third-party file system implementations without the pain. Tech. Rep. CMU-PDL-08-106, Carnegie Mellon University Parallel Data Lab. May.

[2] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. 1986. Mach: A new kernel foundation for UNIX development. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 93–112.

[3] Ahltorp, M., Hrnquist-strand, L., and Westerlund, A. 2000. Porting the Arla file system to Windows NT. In *Workshop on Management and Administration of Distributed Environments*.

[4] Almeida, D. 1999. FIFS: A Framework for Implementing User-Mode File Systems in Windows NT. In *Conference on USENIX Windows NT Symposium*. USENIX Association, Berkeley, CA, 13–13.

[5] Amit Shah. 2009. Feature: High Memory In The Linux Kernel — KernelTrap. `http://kerneltrap.org/node/2450`.

[6] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. 2003. Xen and the art of virtualization. In *Symposium on Operating Systems Principles*. ACM Press, New York, NY, 164–177.

[7] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. 1991. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems 9,* 2, 175–198.

[8] Bershad, B. N. and Pinkerton, C. B. 1988. Watchdogs: Extending the UNIX File System. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 267–275.

[9] Black, R., Barham, P. T., Donnelly, A., and Stratford, N. 1997. Protocol Implementation in a Vertically Structured Operating System. In *IEEE Conference on Local Computer Networks*. IEEE Computer Society, Washington, DC, 179–188.

[10] Borden, T. L., Hennessy, J. P., and Rymarczyk, J. W. 1989. Multiple operating systems on one processor complex. *IBM Systems Journal 28,* 1, 104–123.

[11] Braam, P. J., Callahan, M. J., Satyanarayanan, M., and Schnieder, M. 1999. Porting the Coda File System to Windows. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 30–30.

[12] Brashear, D. Personal communication. 2008.

[13] Bugnion, E., Devine, S., Govil, K., and Rosenblum, M. 1997. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems 15,* 4, 412–447.

[14] Burtsev, A., Srinivasan, K., Radhakrishnan, P., Bairavasundaram, L. N., Voruganti, K., , and Goodson, G. R. 2009. Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA.

[15] Callaghan, B. and Lyon, T. 1989. The automounter. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 43–51.

[16] Carns, P. H., Ligon III, W. B., Ross, R. B., and Thakur, R. 2000. PVFS: A Parallel File System for Linux Clusters. In *Annual Linux Showcase and Conference*. USENIX Association, Atlanta, GA, 317–327.

[17] Cate, V. 1992. Alex-A global file system. In *USENIX File System Workshop*. USENIX Association, Berkeley, CA.

[18] Chase, J. S., Levy, H. M., Feeley, M. J., and Lazowska, E. D. 1994. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems 12,* 4, 271–307.

[19] Chen, S., Falsafi, B., Gibbons, P. B., Kozuch, M., Mowry, T. C., Teodorescu, R., Ailamaki, A., Fix, L., Ganger, G. R., Lin, B., and Schlosser, S. W. 2006. Log-based architectures for general-purpose monitoring of deployed code. In *Workshop on Architectural and System Support for Improving Software Dependability*. ACM Press, New York, NY, 63–65.

[20] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. 2005. Live migration of virtual machines. In *Symposium on Networked Systems Design and Implementation*. USENIX Association, Berkeley, CA, 273–286.

[21] Clements, P. and Northrop, L. 2001. *Software product lines: practices and patterns*. Addison-Wesley, Boston, MA.

[22] Creasy, B. 1981. The origin of the vm/370 time-sharing system. *IBM Systems Journal 25,* 5, 483–490.

[23] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. 2008. Remus: high availability via asynchronous virtual machine replication. In *Symposium on Networked Systems Design and Implementation*. USENIX Association, Berkeley, CA, 161–174.

[24] David A. Wheeler. 2009. SLOCCount. `http://www.dwheeler.com/sloccount`.

[25] Dean, R. W. and Armand, F. 1992. Data Movement in Kernelized Systems. In *Workshop on Micro-Kernels and Other Kernel Architectures*. USENIX Association, 243–261.

[26] DeBergalis, M., Corbett, P., Kleiman, S., Lent, A., Noveck, D., Talpey, T., and Wittle, M. 2003. The Direct Access File System. In *Conference on File and Storage Technologies*. USENIX Association, 175–188.

[27] Don Capps and William Norcott. 2009. IOzone. `http://www.iozone.org`.

[28] Douglis, F. and Ousterhout, J. K. 1991. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience 21,* 8, 757–785.

[29] Ebling, M., Mummert, L., and Steere, D. 1994. Overcoming the Network Bottleneck in Mobile Computing. In *Workshop on Mobile Computing Systems and Applications*. IEEE Computer Society, Washington, DC.

[30] Eifeldt, H. 1997. POSIX: a developer's view of standards. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 24–24.

[31] Eisler, M., Corbett, P., Kazar, M., Nydick, D. S., and Wagner, C. 2007. Data ONTAP GX: a scalable storage cluster. In *Conference on File and Storage Technologies*. USENIX Association, Berkeley, CA, 23–23.

[32] Fabian, P., Palmer, J., Richardson, J., Bowman, M., Brett, P., Knauerhase, R., Sedayao, J., Vicente, J., Koh, C.-C., and Rungta, S. August 10, 2006. Virtualization in the Enterprise. *Intel Technology Journal 10,* 3, 227–242.

[33] Feitelson, D. G. and Rudolph, L. 1992. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing 16*, 306–318.

[34] Fraser, K., Hand, S., Neugebauer, R., Pratt, I., Warfield, A., and Williamson, M. 2004. Reconstructing I/O. Tech. rep., University of Cambridge, Computer Laboratory. August.

[35] FUSE. 2009. FUSE: filesystem in userspace. `http://fuse.sourceforge.net`.

[36] Gartner, Inc. 2009. Gartner Says Virtualization Will Be the Highest-Impact Trend in Infrastructure and Operations Market Through 2012. `http://www.gartner.com/it/page.jsp?id=638207`.

[37] Gingell, R. A., Moran, J. P., and Shannon, W. A. 1987. Virtual Memory Architecture in SunOS. In *USENIX Summer Conference*. USENIX Association, Berkeley, CA, 81–94.

[38] Goldberg, R. 1974. Survey of Virtual Machine Research. *Computer 7,* 6, 34–45.

[39] Gschwind, M. K. 1994. Ftp access as a user-defined file system. *SIGOPS Operating Systems Review 28,* 2, 73–80.

[40] Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A. 2006. Enforcing performance isolation across virtual machines in Xen. In *International Conference on Middleware*. Springer-Verlag New York, Inc., New York, NY, 342–362.

[41] Hammond, L., Nayfeh, B. A., and Olukotun, K. 1997. A single-chip multiprocessor. *Computer 30,* 9, 79–85.

[42] Hansen, P. B. 1970. The Nucleus of a Multiprogramming System. *Communications of the ACM 13,* 4, 238–241.

[43] Herlihy, M. and Moss, J. E. B. 1993. Transactional memory: architectural support for lock-free data structures. In *International Symposium on Computer Architecture*. ACM Press, New York, NY, 289–300.

[44] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. 1988.

Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6,* 1, 51–81.

[45] Huang, W., Liu, J., Abali, B., and Panda, D. K. 2006. A case for high performance computing with virtual machines. In *International Booktitle on Supercomputing*. ACM Press, New York, NY, 125–134.

[46] IDC. 2008a. Server Virtualization Now Firmly Embedded in European Organizations. `http://www.idc.com/getdoc.jsp?containerId=prUK21327108`.

[47] IDC. 2008b. Virtualization Continues to See Strong Growth in Second Quarter. `http://www.idc.com/getdoc.jsp;jsessionid=FT0ISDWWAPJ4SCQJAFDCFFAKBEAVAIWD?containerId=prUS21473108`.

[48] Kantee, A. 2007. puffs - Pass-to-Userspace Framework File System. In *Asia BSD Conference*.

[49] Kantee, A. 2009. Rump File Systems: Kernel Code Reborn. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA.

[50] Karels, M. and McKusick, M. 1986. Towards a Compatible Filesystem Interface. In *Proceedings of the European UNIX Users Group Meeting*. 481–496.

[51] Katcher, J. 1997. PostMark: A New File System Benchmark. Tech. Rep. TR3022, Network Appliance. October.

[52] Kleiman, S. R. 1986. Vnodes: an architecture for multiple file system types in Sun Unix. In *USENIX Summer Conference*. USENIX Association, Berkeley, CA, 238–247.

[53] Lampson, B. W. 1984. Hints for Computer System Design. *IEEE Software 1,* 1, 11–28.

[54] Lang, S. and Ross, R. Personal communication. 2008.

[55] Larus, J. and Rajwar, R. 2007. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers.

[56] Leslie, I. M., Mcauley, D., Black, R., Roscoe, T., Barham, P. T., Evers, D., Fairbairns, R., and Hyden, E. 1996. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal of Selected Areas in Communications 14,* 7, 1280–1297.

[57] LeVasseur, J., Uhlig, V., Stoess, J., and Gotz, S. 2004. Unmodified device driver reuse and improved system dependability via virtual machines. In *Symposium on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, 17–30.

[58] Mark Williamson. 2009. XenFS. `http://wiki.xensource.com/xenwiki/XenFS`.

[59] Matthews, J. N., Herne, J. J., Deshane, T. M., Jablonski, P. A., Cherian, L. R., and McCabe, M. T. 2005. Data Protection and Rapid Recovery From Attack With A Virtual Private File Server and Virtual Machine Appliances. In *International Conference on Communication, Network and Information Security*. IASTED, Calgary, AB, 170–181.

[60] Mauro, J. and McDougall, R. 2006. *Solaris Internals (2nd Edition)*. Prentice Hall, Upper Saddle River, NJ.

[61] Mazieres, D. 2001. A toolkit for user-level file systems. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA.

[62] McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. 1996. *The design and implementation of the 4.4BSD operating system*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA.

[63] McLaughlin, L. January, 2008. Virtualization in the Enterprise Survey: Your Virtualized State in 2008. *CIO Magazine*.

[64] Megiddo, N. and Modha, D. S. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Conference on File and Storage Technologies*. USENIX Association, Berkeley, CA, 115–130.

[65] Mergen, M. F., Uhlig, V., Krieger, O., and Xenidis, J. 2006. Virtualization for high-performance computing. *SIGOPS Operating Systems Review 40,* 2, 8–11.

[66] Michael, M. M. and Scott, M. L. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Symposium on Principles of distributed computing*. ACM Press, New York, NY, 267–275.

[67] Neuman, B. C. and Ts'o, T. Sep. 1994. Kerberos: an authentication service for computer networks. *IEEE Communications 32,* 9, 33–38.

[68] Nightingale, E. B., Peek, D., Chen, P. M., and Flinn, J. 2008. Parallelizing security checks on commodity hardware. In *Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York, NY, 308–318.

[69] Nowicki, B. 1989. NFS: Network File System Protocol specification. RFC 1094, Sun Microsystems, Inc. `http://www.ietf.org/rfc/rfc1094.txt`; accessed May 2009.

[70] Olukotun, K. and Hammond, L. 2005. The Future of Microprocessors. *Queue 3,* 7, 26–29.

[71] OpenAFS. 2009. OpenAFS repository. `http://www.openafs.org/frameset/cgi-bin/cvsweb.cgi/openafs/`.

[72] Ousterhout, J. K. 1982. Scheduling techniques for concurrent systems. In *Conference on Distributed Systems*. IEEE Computer Society, Washington, DC, 22–30.

[73] Padioleau, Y., Lawall, J., Hansen, R. R., and Muller, G. 2008. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Eurosys*. ACM Press, New York, NY, 247–260.

[74] Padioleau, Y., Lawall, J. L., and Muller, G. 2006. Understanding collateral evolution in linux device drivers. In *EuroSys*. ACM Press, New York, NY, 59–71.

[75] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. 1995. Informed prefetching and caching. In *Symposium on Operating Systems Principles*. ACM Press, New York, NY, 79–95.

[76] Pfaff, B. 2007. Improving Virtual Hardware Interfaces. Ph.D. thesis, Stanford.

[77] RedHat. 2004. Bug 111656: In 2.4.20.-20.7 memory module, rebalance_laundry_zone() does not respect gfp_mask GFP_NOFS. `https://bugzilla.redhat.com/show_bug.cgi?id=111656`.

[78] Rifkin, A., Forbes, M., Hamilton, R., Sabrio, M., Shah, S., and Yueh, K. 1986. RFS Architectural Overview. In *USENIX Summer Conference*. USENIX Association, Berkeley, CA, 248–259.

[79] Rimer, M. S. 1999. The Secure File System Under Windows NT. M.S. thesis, Massachusetts Institute of Technology.

[80] Rodriguez, R., Koehler, M., and Hyde, R. 1986. The Generic File System. In *USENIX Summer Conference*. USENIX Association, Berkeley, CA, 260–269.

[81] Rosenblum, M. 2004. The Reincarnation of Virtual Machines. *Queue 2,* 5, 34–40.

[82] Rosenblum, M. and Ousterhout, J. K. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10,* 1, 26–52.

[83] Ross, R. June 2008. Personal communication. Argonne National Laboratory.

[84] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. 1985. Design and implementation of the Sun Network Filesystem.

In *USENIX Summer Conference*. USENIX Association, Berkeley, CA, 119–130.

[85] Sanghi, K. 2009. Implementing File System Virtual Appliances in NetBSD. M.S. thesis, Carnegie Mellon University.

[86] Sapuntzakis, C., Brumley, D., Chandra, R., Zeldovich, N., Chow, J., Lam, M. S., and Rosenblum, M. 2003. Virtual Appliances for Deploying and Maintaining Software. In *LISA: Conference on System administration*. USENIX Association, Berkeley, CA, 181–194.

[87] Sapuntzakis, C. and Lam, M. S. 2003. Virtual appliances in the collective: a road to hassle-free computing. In *HotOS*. USENIX Association, Berkeley, CA, 10–10.

[88] Schmuck, F. and Haskin, R. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Conference on File and Storage Technologies*. USENIX Association, Berkeley, CA, 19.

[89] Seltzer, M., Bostic, K., Mckusick, M. K., and Staelin, C. 1993. An implementation of a log-structured file system for UNIX. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 1–18.

[90] Shah, S. 2009. Porting File System Virtual Appliances to VMware. M.S. thesis, Carnegie Mellon University.

[91] Silvers, C. 2000. UBC: an efficient unified I/O and memory caching subsystem for NetBSD. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 54–54.

[92] Skinner, G. C. and Wong, T. K. 1993. "Stacking" Vnodes: a progress report. In *USENIX Annual Technical Conference*. USENIX Association, 161–174.

[93] Smith, J. E. and Nair, R. 2005. The Architecture of Virtual Machines. *Computer 38,* 5, 32–38.

[94] Srinivasan, R. 1995. XDR: External Data Representation Standard. RFC 1832, Sun Microsystems. `http://www.ietf.org/rfc/rfc1832.txt`; accessed July 2009.

[95] Stevenson, J. M. and Julin, D. P. 1995. Mach-US: UNIX on generic OS object servers. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 10–10.

[96] Sutter, H. 2005. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal 30,* 3.

[97] Sutter, H. and Larus, J. 2005. Software and the Concurrency Revolution. *Queue 3,* 7, 54–62.

[98] Swift, M. M., Bershad, B. N., and Levy, H. M. 2005. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems 23,* 1, 77–110.

[99] Thekkath, C. A., Wilkes, J., and Lazowska, E. D. 1994. Techniques for file system simulation. *Software—Practice & Experience 24,* 11, 981–999.

[100] Verghese, B., Guptag, A., and Rosenblum, M. 1998. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York, NY, 181–192.

[101] VMWare. 2009a. Using Shared Folders. `http://www.vmware.com/support/ws5/doc/ws_running_shared_folders.html`.

[102] VMWare. 2009b. VMware Workstation, Run Multiple OS, Desktop Operating Systems, Virtual PC. `http://www.vmware.com/products/ws`.

[103] Waldspurger, C. 2002. Memory resource management in VMware ESX server. In *Symposium on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, 181–194.

[104] Warfield, A., Hand, S., Fraser, K., and Deegan, T. 2005. Facilitating the development of soft devices. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 22–22.

[105] Watson, A., Benn, P., and Yoder, A. G. 2001. Multiprotocol Data Access: NFS, CIFS, and HTTP. Tech. rep., Network Appliance. September.

[106] Webber, N. 1993. Operating system support for portable filesystem extensions. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 219–228.

[107] Weinhold, C. and Härtig, H. 2008. VPFS: building a virtual private file system with a small trusted computing base. In *Eurosys*. ACM Press, New York, NY, 81–93.

[108] Welch, B., Unangst, M., Abbasi, Z., Gibson, G., Mueller, B., Small, J., Zelenka, J., and Zhou, B. 2008. Scalable performance of the Panasas parallel file system. In *Conference on File and Storage Technologies*. USENIX Association, Berkeley, CA, 1–17.

[109] Welch, B. B. and Ousterhout, J. K. 1989. Pseudo-file-systems. Tech. Rep. UCB/CSD-89-499, EECS Department, University of CA, Berkeley. Apr.

[110] Yang, J., Sar, C., Twohey, P., Cadar, C., and Engler, D. 2006. Automatically generating malicious disks using symbolic execution. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, 243–257.

[111] Zadok, E. and Nieh, J. 2000. FiST: A language for stackable file systems. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 55–70.

[112] Zhao, X., Prakash, A., Noble, B., and Borders, K. 2006. Improving Distributed File System Performance in Virtual Machine Environments. Tech. Rep. CSE-TR-526-06, University of Michigan. September.